**MASTER IN COMPUTER SCIENCE**

# Motivation and global design of a plug-in framework architecture for a medium-large software package

Janmart, M

*Award date:*
2012

*Awarding institution:*
University of Namur


# Motivation and global design of a plug-in framework architecture for a medium-large software package

Maxime Janmart

Promotor: Jean-Noel Colin

**Abstract**

This paper gives a synthesis over the interests of plug-in implementation and the security issues related to such programming. we first select three plug-in frameworks and make a description of them. Then based on what we know, we select the one that interests us the most. Then we present a reference application for this thesis on which we study the possibility of a plug-in conversion. We then describe several ways of implementing this plug-in conversion with the selected plug-in framework. Finally, we describe some security issues and give three high level solutions for these issues.

**Résumé**

Ce document offre une synthèse sur l'implémentation plug-in et les problèmes de sécurité reliés à ce genre d'implémentation. Nous avons d'abord sélectionné trois environnements de programmation pour plug-in et les avons décrit. Nous avons ensuite sélectionné celui qui nous intéressait le plus par rapport à aux critères que nous nous étions imposés. Ensuite nous présentons une implémentation de référence dont nous étudions la possibilité d'une conversion en architecture plug-in. Nous décrivons plusieurs façons d'implémenter cette conversion sous forme d'application plug-in avec l'environnement de programmation que nous avons sélectionné. Pour finir, nous décrivons des problèmes de sécurité et donnons trois sécurité de haut niveau pour ces problèmes.

# Contents

# Chapter 1

# Introduction

In 1986, Alfred Spector, president of Transarc Corporation, co-authored a paper comparing bridge building to software development. The premise: Bridges are normally built on-time, on-budget, and do not fall down. On the other hand, software never comes in on-time or on-budget. In addition, it always breaks down.[gi95]

This study showed that in the computer science field the vast majority of the projects were failures. They were stored into three groups : The first group was the group of the projects that were achieved on schedule without budget extensions, the second group was the projects that were achieved with an extra budget or time needed (¿150third group was composed of projects that had to be given up during the development.[CLM08]

This shows that 83,8 computer science projects were failure. And this forced people to realize that the computer science field had a lot of shortcomings in methodology. As much as there were technical problems, there were also social shortcomings and lack of communication from the team and from the project leader to the clients.

There was thus an emergence of "good programming practices". But more fundamental than that, there was a new consciousness about the importance of good planning for programs.

Afterwards, we realized that a lot of code could be reused. That's to say, you could use a part of a program already implemented into your own application and you win thus a lot of time by working that way. It is becoming more and more important because we realize that we are often implementing functions that are already widely implemented in many other applications. We must also put in evidence that once a code is designed to be reused in also increase the modularity of the application because every component for a specific purpose or service is located in one place and one place only (by exception to the parts of the service that have to communicate with the core. To recycle code, several methodologies were invented. One of them was the plug-in paradigm.

A plug-in can be viewed as a feature of the program wrapped up in a independent part. That means that if the user decides to remove that part, the feature will no longer work but the rest of program will remain functional. Many developers program this way but there are no clear and well known support for this yet. Not that plug-in frameworks are not used, but the majority of developers ignore that plug-in framework exists and that they can facilitate a lot the developer's and the analyst's job.

And if the job is easier for the computer scientist, they can be more efficient and less expensive which would be a solution to the CHAOS report. In an ideal world, the developers would only have to get the features that are asked by the program and to combine them together instead of coding them from scratch. But we are right now far from that world.

## 1.1 Issues inherent to the re-usability principle

As we identified earlier, the need of software re-usability implies that everyone would try to make pluggable applications which would be reused easily without any problems.

The fact is that some plug-in softwares have no reasons to exist. We will take for example the programs used as examples. There is no point in making a complex plug-in architecture for a calculator who only computes additions and subtractions.

There are also programs that are meant for one use only and that will never be changed. There is no need of plug-in infrastructure for a program which is not meant to be customizable.

But the main problem remains in the documentation. There is a need of guidelines and of a framework to develop which would be plug-in oriented. There are existing solutions available on the market but they aren't well known by a majority of developers. We need a robust and trustworthy plug-in framework in order to develop pluggable applications when we need it.

## 1.2 Motivation and methodology of this Master's thesis

This paper will try to resolve the two problematics stated previously. We will try to explain why plug-in programming is or can be interesting in for applications of medium to large size.

To do so, we will have to refer to an application which access has been given to us by the university of Koblenz. This application is called TURM and will be presented in greater details at the end of the introduction.

Also we will try to focus on recurring security issues with plug-in programming.

To do that we selected several plug-in frameworks to show that we did not directly focus on our first discover. Then for the reader to understand the software, we will describe it generally and in details. As for the selection of frameworks, we will explain several solutions we found to convince the reader that the solution we chose was better for the purpose of this thesis.

For the security issues, we selected a collection of articles found on the ACM and on the IEEE which described recurrent security problems with plug-in frameworks and which proposed solution. Those solutions are often abstract and we will thus only provide the logic behind their solution but not a complete implemented solution.

## 1.3 Structure of this Master's thesis

First as we must develop a plug-in application, we will have to choose a plug-in framework. We browsed the Internet to get some advice about what was available in the area of plug-in frameworks and it came down to three choices : JPF, JSPF and OSGi [oc09]. We will establish criterion based on our client's needs and advices in order to select the best framework for him.

Then we will present Turm which is the application we will transform into a plug-in application. To present it, we will first need an explanation of policy expression languages because Turm is supposed to be a tool for one of this language : ODRL (open digital rights language). As Turm is not a very large software package, we will describe it first as if we were selling it, and then we will go into greater details to explain how all the functionalities are implemented.

That being done, we will identify the parts of Turm which might be interesting to extract as plug-ins and explain which of them we could extract and why. As the program was designed in a monolithic way, some plug-ins were impossible to extract.

Then we will explain how to make a plug-in out of a monolithic architecture and develop three solutions for that. We will explain which solution is the best and what the other are unsatisfiable for this instance.

We will also discuss security issues identified by developers and researchers and how they did manage to solve them.

At last, as a conclusion we will wrap everything up we will give examples of plug-in architectures that work with testimonials of developers which used a plug-in architecture and find it more convenient than monolithic architecture. Of course these developers will be selected because they have scientific arguments and not only because they chose plug-in architecture randomly.

## 1.4 Turm

At last for this introduction, we must introduce Turm briefly. Turm is a software (not yet open-source) which was first developed to be a tool to implement ODRL licenses which are policy expression licenses.

Then as ODRL grew to be nearly a w3c standard, there was a reference implementation needed. As Turm was already nearly implementing policy expression licenses, it was converted into this reference application.

It was assigned to many students and the developers that worked in Turm are not clearly identified. But as its main author Daniel Pähler decided to extend its functionality for it to be some kind of peer to peer client with rights expressiveness, there was a need to extend the media file extensions that Turm could manage and that is where the need of plug-in infrastructure began.

# Chapter 2

# Plug-in Frameworks

## 2.1 Introduction

One of the biggest problems in large software packages is that sometimes you don't need everything that is provided to you. Sometimes you only use half the options available or you only use one specific function because this large software package does it better than the regular application you use. So when you run this large software package, it takes a while, penalizing you because it takes a lot of memory for the small function you are about to use. For instance, as time goes by, Microsoft Word is becoming a more and more complete software. There are a lot of features to grant the user the most comfortable way to edit a text and to format it the way he'll enjoy most. However this features have a cost and on regular computers, Word takes some time to start. But these features aren't used by everyone; there are lessons to take in order to pretend yourself expert in office applications. Thus some people have to bare the hugeness of the latest Word versions even if they only use it to type a text without even formatting it.

An other problem with large software packages is that sometimes you really would like to add a feature to them but you couldn't think of it when you first created the application. Unfortunately the developers who worked on your application quite derived from the class diagram you made, leaving the application a bit hard to understand for you. It is then very complicated to add a feature which wasn't been planned to be added at the beginning. An example of application which succeeded in this area is Mozilla Firefox: there are a lot of plug-ins users can add or remove depending on the way they use Firefox. There are numerous tool bars and little softwares to help them customize their Firefox.

This is the whole interest in plug-in frameworks. Their goal is to make available the loading of modules dynamically. Let's say a developer has an application but only uses half the features. he can disable the half he doesn't need, gaining memory and executing time by the same way. he can also, if an interface is provided, create his own feature for the application and load it dynamically. This is the greatest strength of plug-in application : their modularity.

This chapter will introduce you to some plug-in frameworks. We will only browse the most known ones : Java Plug-in Framework (JPF), Java Simple Plug-in Framework(JSPF) and Open Service Gateway Initiative (OSGI)

We will present the frameworks, give their qualities and shortcomings and try to make a comparison between them in order to select the most suitable for the job we are planning to do. Of course there will be scientific criteria so make our selection but also intuitive criteria. By intuitive we mean all the criteria that would improve the comfort of the user. By scientific criteria we mean all the technical features of each frameworks, in what area they are more suited than an other framework and how they would fit with the particular case of TURM.

We will not though make a list of all the criteria we used. Indeed, it is hard to find a list of criteria, they are more organized in abstract concepts. Also, as the field of plug-in framework is not sparse, there will be no technical analysis of each frameworks so we must rely on what we will find about them. We must also insist on the fact that the descriptions are mostly "commercial descriptions" which will not put in evidence the flaws of the solutions. We will have to find them ourselves.

We will use two point of views to describe the frameworks : one aimed at the developers and one aimed at the performances of the framework.

Those two angles are needed because TURM is not a finished solution: the purpose of plug-in softwares is to allow users to develop and add plug-ins whenever they want and need it. We must thus provide the future developers with a reliable and easy to use framework. Also, as the developers will have to learn how to use the framework, it must be well documented.

But the first angle is not enough as a framework can be very easy to use, that being explained by the lack of completeness. It might be better to use a harder to understand framework with a lot of useful functionalities, than an easy to understand one which can only develop an "hello world" application.

## 2.2 OSGI

### 2.2.1 Motivation

The OSGI was founded in March 1999. The first OSGI specification was created in 2000. Their two main goals were to create a Java platform who could be manipulated remotely and which would allow a full dynamic component model.[IGPK]

They created OSGI because there was no Java default implementation for such systems. The JVM did not support the life cycle of an OSGI bundle. That's to say the JVM does not support a module to be loaded, started, updated, stopped and removed from an application during its execution.

To describe it, we based ourselves on the official OSGi website and on two books [JM10] [dCA11] and a website [com99].

OSGI uses the LEGO principle: that is a set of properties that a framework should have. First property is re-usability. An application or a part of an application developed in OSGI should be transferable from one project to another. That is an useful property to gather features from a well implemented application to your application which needs that feature. That will spare you time and money.

Second is scalability. An application developed in OSGI should be able to manage an increasing charge of the requests. That's to say : an OSGI application shouldn't have a bottleneck.

Third is portability. An application should ignore the underlying hardware and should have a standardized execution environment. This is a quality that all application should have in order to be able to run without any problems on different machines with different operating systems.

Last is the managing of the life-cycle. OSGI should help developers to manage the life-cycle of the application. They should be able to turn down an application part or to load a new part of the application easily while running it.

### 2.2.2 Bundles

One of the main component of an OSGI application is the bundle. In the Java world, a bundle simply represent one or more packages. In OSGI world, a bundle is an application in itself and can be ran independently than the rest of the application.

**Lifecycle description**

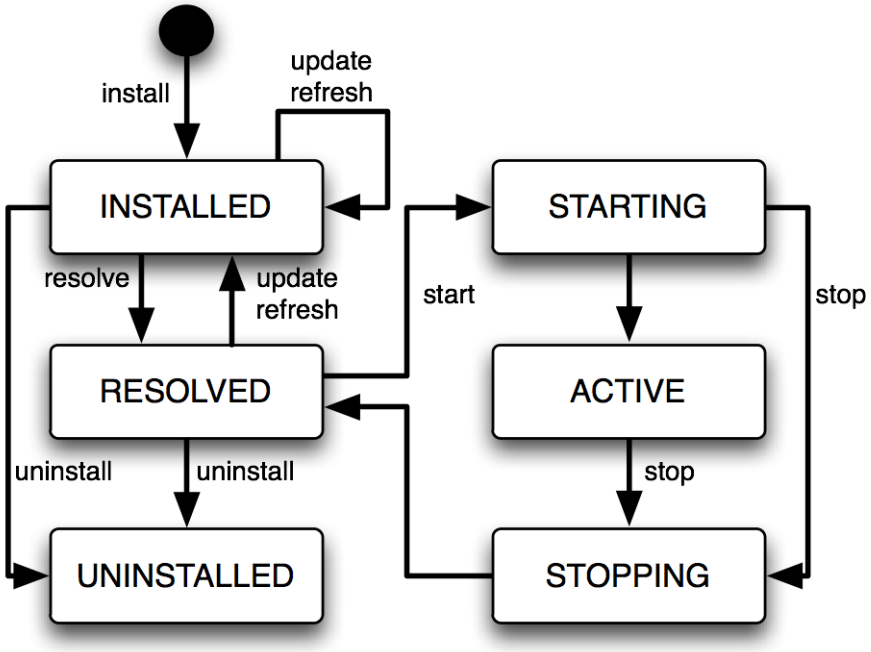


Figure 2.1: The bundle's lifecycle

Before describing what is a bundle, we will just say that it is an application which can interact with other applications.

To begin the life-cycle,a bundle must be created and then installed into the OSGi framework. The possible errors here are that there is a syntax error in the code or that the dependencies aren't possible to satisfy. For example, if there is a cycle in the dependencies. Let's say there are three bundles A B and C. In order to run them, all the bundles on which they depend must be running or available. If A needs B, B needs C and C needs A, none of them will be ready and it will be an impossible problem to fix. We will have to think differently in the creation of the bundles. Once it is installed, then can of course be uninstalled or updated. We will make a more complete scenario at the end of the description.

When a bundle is installed, it can be resolved. In the resolving phase, OSGI tries to get all the dependencies of the bundle and see whether they can run or not. If they are able to run, then the bundle goes to the resolved state. it can of course be uninstalled, but if it is updated, it doesn't stay in the resolved state. In deed, when a bundle is updated, the dependencies are not yet satisfied by the OSGI framework thus there is a need of re resolving the bundle to make sure all the dependencies are matched.

Then once a bundle is resolved, it can be started. When a bundle starts, there is a special class called an Activator which contains an operation (start). This is the equivalent of the Java main class. Until the operation finishes, the bundle remains in the starting state. The only state it can go from there is the stop state. When the bundle's start operation is finished, it is in the active state and thus it is running. When the bundle is stopped, it is the same logic as the start activity: the stop operation from the Activator class is called and until it is finished, the class remain in the stopping state. Once it is stopped, the bundle returns to resolved state.

**Scenario of the life-cycle of a bundle**

Let's imagine a very simple bundle which goal is only to add numbers. The only operation it provides is the add(int i, int j) operation which returns an int. To add a dependency, let's say a new type of integeris created : MyInteger. The operation thus becomes public MyInteger add (MyInteger i, MyInteger j). MyInteger is in another bundle which is permanently running and doesn't need any changes.

the plug-in is installed. As it is a very simple plug-in there won't be any syntax errors. Then it must be resolved. The OSGI framework will look, among others for the bundle containing MyInteger. As it is running, the calculus bundle becomes resolved. When started, as there are very few operations, it goes with no time to the active state.

Everything is running smoothly but to make things more interesting,the calculus bundle should be able to substract things as well. Then a public MyInteger sub(MyInteger i, MyInteger j) operation needs to be added. As the old version of the bundle is running, it cannot yet make subtractions. it must be stopped so that it becomes unreachable for the other bundles. Then an update is performed. It will go back to installed state and will have to be resolved again. As no dependencies are changed, it will become resolved and ready to start again.

### 2.2.3 Services

Having bundles is useful, but in order to have an useful bundle, it is supposed to provide the other bundles operations. This is called a service. In OSGI, a service is an implementation of functionalities wrapped into a bundle. A service can have several functions.

The most used one is to export a functionality from a bundle to other bundles. As we said, functionalities have to be grouped into a bundle and then made available for others. These functionalities can be imported from other bundles which is the dual process of the previous one. listeners or events can be set up from other bundle. Devices can also be accessed by the bundles. For example a printer can be shared with the OSGI framework and the classes that uses your printer can be exported in order to become available for other bundles.

### 2.2.4 The OSGI Manifest.mf file

We stated earlier that we should make bundles available for others but we did not quite say how this does work. As a bundle can be seen as a standalone Java application, each bundle has its own manifest file which gives a lot of information about the bundle. Here is an example.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: TURM Core
Bundle-SymbolicName: TURMCore
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: de.uni_koblenz.aggrimm.turm.service.TurmActivator
Bundle-Vendor: Daniel Pähler, Uni Koblenz
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Export-Package: de.uni_koblenz.aggrimm.turm.core;
  uses:="de.uni_koblenz.aggrimm.turm.facades,
   de.uni_koblenz.aggrimm.turm.media_manager,
   de.uni_koblenz.aggrimm.turm.license_manager,
   de.uni_koblenz.aggrimm.turm.db_manager,
   de.uni_koblenz.aggrimm.tools.configurationmanager",
```

```
 de.uni_koblenz.aggrimm.turm.exceptions,
 de.uni_koblenz.aggrimm.turm.hashCalculator,
 de.uni_koblenz.aggrimm.turm.payloadExtractor
Bundle-ActivationPolicy: lazy
Bundle-ClassPath: .,
 lib/antlr-2.7.6.jar,
 lib/bcprov-jdk14.jar,
 lib/beepcore.jar,
 lib/c3p0-0.9.1.jar,
 lib/commons-cli-1.2-javadoc.jar,
(...)
 lib/sqlitejdbc-v056.jar,
 lib/sunjce_provider.jar
Import-Package: org.osgi.framework;version="1.5.0",
 org.osgi.util.tracker
```

The first is the manifest version. Its signification is trivial.

The second is the bundle manifest version. This should be updated each time we change something in the bundle. For example we could add a name to the Bundle-Vendor variable, add or remove dependencies, etc.

The third is the name of the bundle in the OSGI framework. We will see later how to manage OSGI with equinox and with a console tool. This is the public name of the server.

In opposition to that, we have the symbolic name of the bundle who should be unique in the OSGI framework.

Then, same as the manifest version is the bundle version. This work like an application version, it is advised to use the last digit for small updates like bug corrections, medium digits for the noticeable modification like a modification of a feature and the first digit for big modifications. For example a version 2.1.12 should mean that from the beginning there has been one huge modification, after this huge modification there was a noticeable modification done and in this noticeable modification 12 small fixes have been done.

The bundle activator variable is set to the class path of the activator. It is not forced to be called "Activator" but has to be referenced in the manifest file.

The bundle vendor is just an informative variable to identify the owner of the bundle. Note that it is based on honesty because there are absolutely no protections to ensure that the owner set in the manifest file is the real owner. This can be modified even if the Jar file is already made. We could grant the security of the bundles if there was a way to protect it against post exportation modifications.

The required execution environment is generally left at its default value set by the OSGI framework. It is the version of Java this bundle was developed on and so it might not be compatible with previous versions.

The export-package is the variable that renders packages of the bundle available. Let's say there is a calculator bundle with two packages : add and sub which are package related to addition and subtraction operations. If add only is put in the export package, the bundles which will access to the calculator bundle won't be able to access subtraction operations. Even if they have the complete bundle, OSGI will prevent them from using a package which is not in the Export-Package variable.

The bundle activation policy describe how the bundle should be started. Actually the only supported value is lazy. That means that the bundle's activator won't be started until a call from this bundle's class is made. An other value could be for example auto which would automatically start the bundle when it is resolved even if there are no calls at all during the execution of the application.

The bundle classpath sets all the imports that must be done. As this manifest is the Turm core's manifest, there are a lot of imports and we just cut them from the file.

The import-package value is the dual of the export one. It tells OSGI which bundles are needed by the application to function correctly. The only mandatory import package is org.osgi.framework;version="X.X.X" because the OSGI version used has to be specified. In Turm, we had to add another package, the tracker which will be presented in the next section.

### 2.2.5 Service registration and Activators

So far we have created a bundle, we have made the manifest file specifying which packages of other bundles were needed and which packages the other bundle could access in our bundle. Now we are going to go into greater details in the internal functions of OSGI. The goal of this thesis is not to create OSGI experts but to give enough knowledge to understand how everything is done.

We will first give an example of simple activator.

```
package de.uni_koblenz.aggrimm.turm.oldplug-ins;

import java.util.Dictionary;
import java.util.Hashtable;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

import de.uni_koblenz.aggrimm.turm.payloadExtractor.PayloadExtractor;

public class PayloadExtractorMP3Activator implements BundleActivator {

private ServiceRegistration sr;

@Override
public void start(BundleContext context) throws Exception {
// Initialization of payloadExtractorRegisterService
PayloadExtractorMP3 pemp3 = new PayloadExtractorMP3();
/*creation of the filter that will be used by the service tracker
 * iface is the field used to define the name of the interface or
 * abstract class of the class you want to register.
 */
Dictionary<String,String> d = new Hashtable<String, String>();
d.put("iface", "PayloadExtractor");
sr = context.registerService(PayloadExtractor.class
.getName(), pemp3, d);
System.out.println("PayloadExtractorMP3 is registered");
}

@Override
public void stop(BundleContext context) throws Exception {
sr.unregister();
System.out.println("PayloadExtractorMP3 is unregistered");
}

}
```

The firsts lines define the imports. We will justify them while browsing through the code. Of course the class whatever its name, has to implement the bundleActivator interface of OSGI which justifies the first OSGI input. As a service is created,one of its class had to be available to the OSGI framework. In this example we instantiate a payloadExtractor for MP3 format.

We will ignore the dictionary for the moment and focus on the sr variable. It is the attribute used to register the service through OSGI. This is the most basic way to do this. The developer must provide to the registerService operation the name of the class you want to share, an instance of the class and a third value who can be set to null for the moment. We will see at the end of the section a cleaner way to implement this.

Now our service is registered, that means that OSGI is aware that a payloadExtractor for MP3 files is available on the framework. But we still do not know how to access it.

The last thing we should mention is the BundleContext variable: it is a variable which allows the bundle to interact with the framework. So this variable contains, among others,

a way to retrieve the service published earlier.

### 2.2.6 Accessing a service : static way VS service tracker

```
public class TurmActivator implements BundleActivator {
    ServiceReference payloadExtractorMP3ServiceReference;

    public void start(BundleContext context) throws Exception {
       payloadExtractorMP3ServiceReference = context.
       getServiceReference(PayloadExtractorMP3.class.getName());
       PayloadExtractor pemp3 =(PayloadExtractorMP3)context.
      getService(payloadExtractorMP3ServiceReference);
      *business code*

    }
    public void stop(BundleContext context) throws Exception {
        System.out.println("Goodbye World!!");
        context.ungetService(payloadExtractorMP3ServiceReference);
    }
}
```

This is an easy and quite unclean way of registering a service. There is an attribute to retrieve the service's reference which is gotten from the bundle context. The problem we will find to this is that these operations will have to be done each time you need a service or global attributes to manage this and instantiate them at the right time will be needed. This implementation could work in a majority of situations, but the context.getService will only return one object which matches the references you input.

A much cleaner way to proceed is to use the service tracker. This is where the dictionary you put in the place of the null parameter of the service activator comes in handy. This dictionary is a way to tell the service tracker what kind of service it is. The ServiceTracker is a class that browse the OSGI framework for services and activates them if they match a certain filter.

So instead of using operations in the activator, the service tracker is instantiated. This is the way Turm's plug-in management was designed.

```
package de.uni_koblenz.aggrimm.turm.service;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

import de.uni_koblenz.aggrimm.turm.core.Turm;
import de.uni_koblenz.aggrimm.turm.gui.controller.GUIHandler;

/**
 * This class provides the connection point needed by OSGI when installing Turm
 * as a bundle (it is therefore also set in the MANIFEST.MF file).
 *
 * At the moment, it also calls the TurmThread when started, which eventually
 * leads to some "try-out" code from Main being executed.
 *
 * Finally it instanciates the service tracker in order to be able to register
```

```
 * the services
 *
 * @author tulkas, mjanmart
 */
public class TurmActivator implements BundleActivator {

private TurmServiceTracker tft;

public void start(BundleContext context) throws Exception {
System.out.println("TurmActivator started");
// Initialization of TURM
Turm turm;

turm = Turm.getInstance();
// start the gui
new GUIHandler(turm.getFacadeFactory());
tft = new TurmServiceTracker(context, null);
tft.open();

}


public void stop(BundleContext context) throws Exception {
System.out.println("TurmActivator stopped");
tft.close();
}

}
```

All the business code is in the TurmServiceTracker. Another asset is that the service tracker only needs to be instantiated once for all the services. With 50 services, the operation above would have to be called 50 times. In the service tracker everything is managed more easily and you just have to add a condition to the filter.

```
package de.uni_koblenz.aggrimm.turm.service;

public class TurmServiceTracker extends ServiceTracker {

public TurmServiceTracker(BundleContext context,
ServiceTrackerCustomizer customizer) {
super(context, getTrackerFilter(context), customizer);

}

@Override

public Object addingService(ServiceReference reference) {
Object service = super.addingService(reference);
if (service instanceof PayloadExtractor) {
try {
Turm.getInstance().getPayloadExtractorManager().addTurmFeature(
```

```
(PayloadExtractor) service);
} catch (plug-inException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
} else if (service instanceof HashCalculator) {
try {
Turm.getInstance().getHashCalculatorManager().addTurmFeature(
(HashCalculator) service);
} catch (plug-inException e) {
e.printStackTrace();
}
}
return super.addingService(reference);
}

public static Filter getTrackerFilter(BundleContext context) {

Filter res = null;
try {
res = context.createFilter("(| (iface =PayloadExtractor) (iface =HashCalculator))");
System.out.println(res.toString());
} catch (InvalidSyntaxException e) {
System.out.println("synthax error during creation of filter");
;
}

return res;
}

@Override
public void removedService(ServiceReference reference, Object service) {
if (service instanceof PayloadExtractor) {

Turm.getInstance().getPayloadExtractorManager().removeTurmFeature(
(PayloadExtractor) service);

} else if (service instanceof HashCalculator) {
Turm.getInstance().getHashCalculatorManager().removeTurmFeature(
(HashCalculator) service);
}
super.removedService(reference, service);
}
}
```

The constructor of the servicetracker is made of 3 inputs : The bundle context, a filter and a customizer. The bundle context is used for the service tracker to be able to access the context in which the services will be published. The filter will be explained in the next paragraph. The ServiceTrackerCustomizer is called whenever there is business code to do when a service registers. For instance if a service needs to get the time when he was spotted by the ServiceTracker, the code used will be in a class implementing the

ServiceTrackerCustomizer interface.

The filter is a way to focus on specific service and ignore others. For instance, in Turm, it would have been possible to make a service tracker for each type of service. That's to say making a service tracker for the payload extractors, and one other for all the hash calculators but it would have been too complicated for what we wanted. The filter specifies which services can be taken care of thus leaving a filter empty means that no service will be registered at all.

That is where the dictionary we put in our service comes in handy. This dictionary is used by the filter to sort the services. A basic filter will just take into account predicates under the form "key = value" of the dictionary. There are more interesting ways to use the filter but as it is just a general presentation of OSGI, there is no need to go in further details. So for example if an user wants to register services that implement a certain interface I, he might want to put in his filter (iface =I) assuming that the person who made the service put the name of the interface in the "iface" key of its dictionary.

Using a service tracker requires thus to be more neat when you work and to communicate with others. The first version only cares about the name of the class, the service tracker however needs something to work with its filter. Another interesting thing about the filter is that it uses the reverse polish notation. As a reminder "1+1" is written "+ (1 1)" in reverse polish notation. Every operator is put at the beginning of the line. Some programming languages like Lisp still use this notation. "I want iface to be I1 or to be I2" will be expressed "(| (iface = I1) (iface=I2))" in the filter, "|" being the logical operand for the "or" in logics.

In the next section, we will explain which pre-configured service can be provided by OSGI.[IGPK]

### 2.2.7 System services

#### Logging

The logging service provides logging of information, warnings, debug information or errors. It can be used to debug more easily the application. Indeed, it is much harder for plug-in applications and for distributed applications to be debugged via for example the eclipse debugging system.

#### Configuration Admin

This service provides information about the deployed bundles' configuration. It is also useful when you want to correct configuration errors.

#### Device Access

This service makes Plug and Play easier through OSGI. It helps the the application with automatic detection and attachment of devices.

#### User Admin

This services covers the authentication and authorization issues. It manages a database with the private and public user information.

#### IO Connector

This service implements the CDC/CLDC javax.microedition.io package as a service. This service allows bundles to provide new and alternative protocol schemes[IGPK].

#### Preferences

This service is an alternative implementation of the Java Properties class.

### 2.2.8 Component Runtime

This service provides an XML base declaration of the dependencies of the services. It is useful as the goal here is to be able to unplug and plug new services at any time and this service will show us the dependencies we need to manage.

#### Deployment Admin

This standardizes the access to a subset of the responsibilities of the management agent.

#### Event Admin

This service provide an implementation of the publish and subscribe model for the bundles. This helps inter-bundle communication: a bundle can publish it's service and when a second bundle notice it, it can subscribe for this service as it needs it to do some business code.

#### Application Admin

When you work in a complex environment with many different applications available simultaneously, this service provide a way to manage these application more easily.

### 2.2.9 Protocol services

#### HTTP Service

This allows information transfer from OSGI using HTTP

#### UPnP Device Service

This service contains a specification about how to develop OSGi bundles in order to get interoperability with Universal Plug and Play(UPnP) devices.

#### DMT Admin

This service contains an API from which you can manage a device using the Open Mobile Alliance (OMA) device management specifications.

### 2.2.10 Miscellaneous services

#### Wire Admin

This implements the producer and consumer service connection

#### XML Parser

This service allows a bundle to locate a parser with desired properties and compatibility with JAXP

#### Measurement and State

This service is responsible for all the measurements handling in an OSGi service platform.

### 2.2.11 Conclusions about OSGi

OSGi seems to be an easy-to-use and easy-to-install framework. As a matter of fact it didn't pose any problems when we installed it. Everything worked smoothly and the only difficulty was to understand clearly how the filter of the service tracker worked.

It is also a well documented and well known plug-in framework as there are a lot of websites and book treating the OSGi subject. We must underline that the documentation and the books are not always correct (some examples not being updated from one version of a book to another) but as there are plenty of sources, it is not difficult to find another one.

Last but not least, OSGi is still in progress. It might not seem interesting but as a community is still living then the concept the community works on lives with it. As we will describe further, it was not the case for every plug-in frameworks.

## 2.3 JPF

This section will summarize what is said on the official Java plug-in Framework website[JPF]. All the graphics and descriptions are inspired from this website as there is very few documentation on other websites. The main problem here is that there are very few information about the owners and we are not sure who are the owner. Moreover, the website seems to remain unmaintained as the copyright only extends itself from 2004 to 2007 and had its last modification on July 2007.

We will however describe it in lesser details than OSGi because the documentation is not as furnished as it is with OSGi.

### 2.3.1 General description

As it is shown in figure 2.2, there are three main components for JSP: the plug-in manager, the plug-in registry and the path resolver.

The plug-in registry stores all the meta information about the plug-ins that were discovered in a repository. This contains interfaces in order to abstract the meta informations about the plug-ins. They also contains manifests designed in XML who are attached to them.

The path resolver is the component that resolves the location of the resources that were discovered. This works simply with a mapping of manifests and locations. For example in Linux systems it will match a "home" variable with "/home/user" where in Windows vista and newer systems it will match the variable with "C:\Users\Foo\Documents". In order to keep the local resources available, shadow copies are made. These shadow copies are copies from the plug-in resources that are in use so you don't lock them while trying to resolve them.

The plug-in manager as its name states is responsible for plug-in management. It loads and activates the plug-ins. This manager brings together the two other components in order to have one single point to start JPF. This can be seen as a good idea because one thing only has to be started, but it is also a bad idea because there could be there a single point of failure.

A typical scenario would be to use a boot launcher, which would initializes all the registries, instantiate the plug-in manager and then activate the main plug-in. By activate the main plug-in we mean using the main operations which would be the Activator's operations of an OSGi bundle.

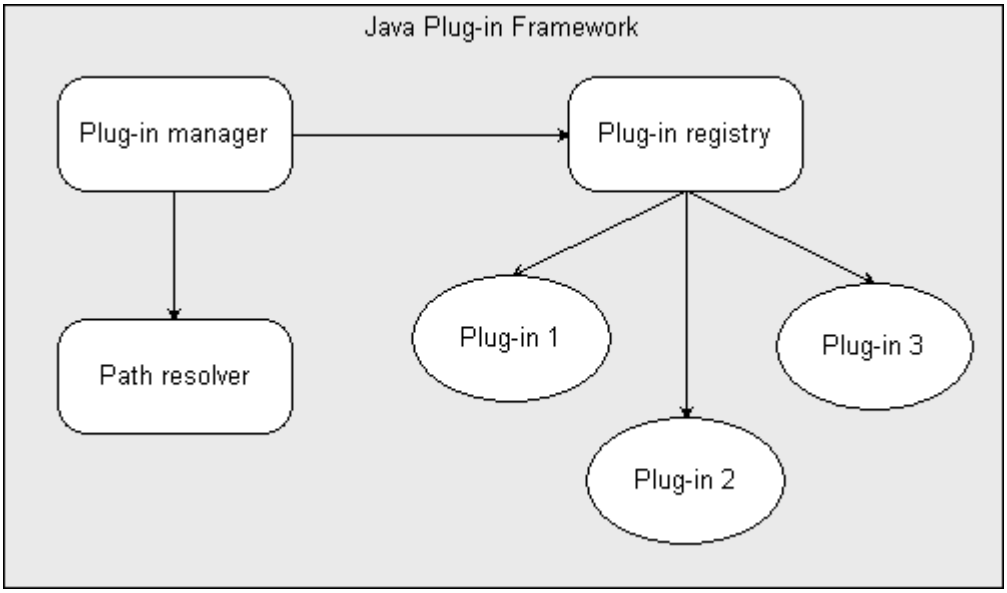


Figure 2.2: JPF architecture

## 2.4 JPF Boot diagram

The table 2.3 explains the guidelines to follow in order to start an application. This is somehow a formalized diagram about how to JPF work at startup for an application.

First the configuration is loaded. This step shouldn't need further explanations. Then the error handler must be instantiated in order to manage anything that can go wrong at startup and during the application. Next, JPF splash screen is shown.

This is where the real application initialization starts. First the logging system is being configured in order to log everything that will happen in the application. Then JPF analyzes what plug-ins are needed for the application to work. Next the plug-in manager discussed above is instantiated. First thing to do then is to check the integrity of the plug-ins by resolving their paths. The application plug-in which is the "core" of the application is located and initialized.

Once the application is initialized, all that is left to do is to start it and then make it run until it has to be stopped.

We stay generic and don't go into details because there is very few documentation and we would not like to make any guesses that would be wrong towards what is really underneath in the "low-level" layers. On the JPF website is located a tutorial section in which they describe a demo program. This didn't look educational at all and will not be discussed about in this thesis in order to not confuse the reader.

The last thing left to talk about are the tools which could be interpreted the same way as the OSGi services.

### 2.4.1 JPF Tools Reference

The tools are implemented like ants files with parameters. We will browse them in a verbose way. We will not make a man page but we will give a feeling about how this all work.

#### Integrity tool

This tool is made to check the integrity of a plug-in collection The main options here are includes or exclude functions: these operations can be performed with one file or a list of files. A verbose mode is also available to get more information about the data integrity.

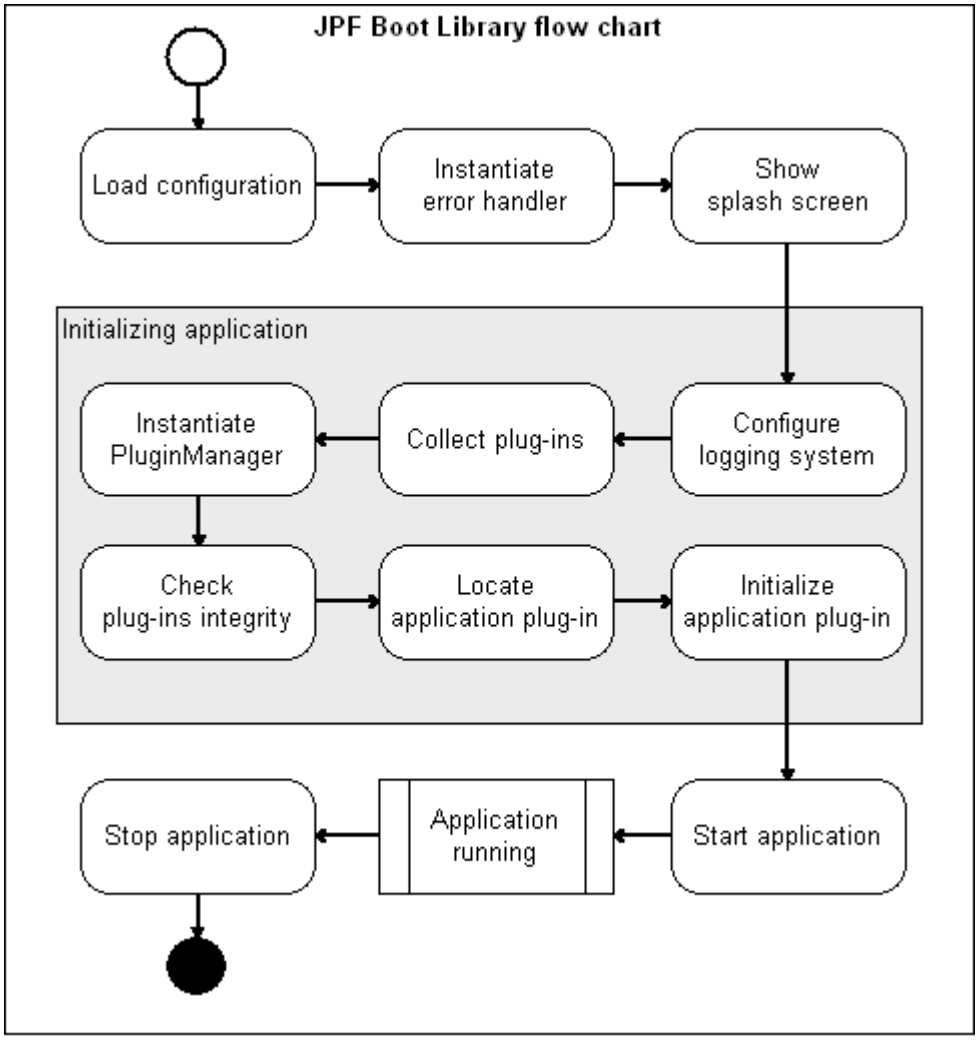


Figure 2.3: JPF boot library flow chart

### Documentation tool

The documentation tool generates the documentation for the plug-ins. This tool contains the same options as above with additional options about the location of the documentation, the CSS style sheet to use, the encoding and template files.

### Plug-in archive tool

The plug-in archive tool is as a matter of fact a multiple tool as there are options to zip plug-ins and to unzip them. These options are roughly the same as the integrity tool operations the only difference for archiving tasks is that you can specify the destination file. For the un-archiving tasks, the name of the source file can be specified as well as the destination files where it will be extracted.

### Single file plug-in tool

The single file plug-in tool is a tool which allows the user to manage the plug-ins separately. Instead of zipping a whole application several parts can be zipped separately, versionning can be made and so on. The option are roughly the same as the integrity tool options.

#### Manifest info tool

This tool is the first that doesn't have the same option as the first tool. It is designed to read plug-in manifests and store them into project properties. The manifest to be read has to be specified. the other options are made to organize the properties file. An id, the version, the vendor can be set. The id, the version and the matching rule of the plug-ins can also be set. This last option is only used for fragments if you decided to fragment your plug-ins.

#### Version update tool

This tool is responsible for versionning. It will upgrade the plug-in version when it is updated. The option here are the same as the integrity tool options except that there is a timestamp value which will define whether the version will contain the timestamp or not.

#### Classpath tool

The classpath tool is used to set the classpath of the application (for instance where the libraries are located). This tool also has the same options as the others. It becomes a bit unclear because some of the values are coupled and one of them only should be instantiated.

#### Sorting tool

The sorting tool is the tool with which you will be able to set an order to the plug-in execution.

### 2.4.2 Conclusion about JPF

JPF was a bit disappointing. First there is no clear installation guide only a tutorial and some vague information on the website. The only thing we found was a tutorial stating that eclipse Helios was the "newest" version back then. And even that tutorial is not very clear.

Also during the installation tryouts, we realized that it was impossible to have OSGI and JPF on the same eclipse because they were mutually exclusive. When trying to install JPF on a different eclipse, we realized that JPF anyway needed some of the OSGI classes.

The JPF community seems dead as the last update of the official website seems to be on July 2007.

We could not install JPF completely and this is why we couldn't appropriate ourselves the framework. The lack of structure of this section is mainly due to the lack of structure of JPF's main website.

## 2.5 JSPF

### 2.5.1 General Description

Java Simple plug-in Framework is as its name let wonder: a simple plug-in framework. This meaning it is something to be used when there is a need of quick application for a reunion or a demonstration.

Even if the names might suggest it, there is no relation between JSPF and JPF.

There is not a lot of information about JSPF about the low-level structure.

There are no installers because JSPF is more a library than a framework. Indeed, it is possible to develop plug-in applications with JSPF but there are no support for it. As a comparison, OSGi can be developed with eclipse, but there are also other frameworks that are designed especially for OSGi: it would be unproductive to make Java code with them

but they possess a command line tool which is easy to use and which helps you manage the plug-ins.

JSPF however allow users to manage plug-ins with Java classes. It is heavily annotation based: everything you specify in the annotations are meant to help the management of plug-ins. For example it can be specified on a class that it is a plug-in implementation that this operation requires a certain type of plug-in, etc.

The use that OSGI does of manifest file is made here through the annotations.

There are no further information except from the javadoc.

### 2.5.2 conclusion

This plug-in framework is sitting on the fence between OSGi and JPF. The reason is that it doesn't present itself as a great software for huge software packages. It is more described as a tool.

Even if it might seem alive, the community is only composed of 4 persons and 76 users spread among the world. To make an even comparison, OSGi possesses more groups than JSPF possesses developers.

It has a tutorial which describe clearly how to design an hello world application in a video and as JSPF only has to be included as a library, you only need to download the files and no installation is needed.

## 2.6 OSGi vs JPF vs JSPF

### 2.6.1 Documentation and support

As we saw previously, OSGi is far more documented that both others plug-in solutions. The community is also much larger in OSGi than in both the other solutions even if JSPF seemed to have reactive developers.

It is important to have a good documentation and a reactive support for our choice. We must not forget that even if we do implement a plug-in solution and discuss its use, one of the constraints given by our client was an "easy to re-use plug-in solution". By that we mean that other students will write thesis and might need to implement a plug-in for TURM. This means that they must have a reliable plug-in framework because they can have any degree (some parts of TURM were developed or designed by bachelor students).

OSGi is well documented: there are books, a support, and websites where tutorials were made. If a student is ever stuck, he will have a lot of documentation easily reachable. Regarding JPF and JSPF, the only documentation we found without searching too deep are posts with very specified issues which led to a single solution and which is not enough to understand the language. The tutorials made for JPF were just all installed program you just watch running. JSPF was a bit better specified as there is a video of a user implementing a "hello world" plug-in. It is though insufficient to understand the whole language easily.

On the other hand, having a lot of documentation also implies having documentation which is partially or totally wrong. The user will have to be careful about the documentation he follows.

Regarding the support, JPF seems to be dead already so it might not be a good choice to take it unless it is considered completely finished and working. But even with a finished application, there will always be patches to make to match the latest version of the programming language. As this is a Java application we shouldn't have to worry about operating system evolution though.

JSPF does have a reactive support, but it might not be able to handle a growth of the questions. As there are few users, there is few support needed and there are few bugs that

are discovered. If JSPF become more popular, the support community might not grow as fast as the users. There is not a lot of documentation so there won't be a lot of experts in that field neither.

### 2.6.2 Deployment of the framework and plug-in development and management

As said earlier, JSPF was the easiest to deploy as you only have to integrate a library to your code. The problem with that is that by opposition to OSGI, the feeling that a framework really works behind your code is less present. When a developer implement a plug-in software, he can't manage the plug-ins as easily as with OSGi: he has to implement himself a plug-in management solution for your application where OSGi already provides several: for instance equinox which is an eclipse plug-in allowing the plug-in management in the eclipse framework or felix which is the opposite and which is a command-line tool which allows users to manage their plug-ins.

We were unable to install JPF easily and figured out that if a program is not well documented enough to make its installation easy, it would be a sufficient reason to not use it. The manifest files used by JPF to manage the plug-ins like OSGi does is a good idea though, but when an unexperienced reader compares the manifest files from JPF with the ones from OSGi, the OSGi manifest files seem far easier to understand.

Also OSGi has a great advantage on the two others because it clearly provides an easy to use service tracker. There may be similar ways to work with JPF and JSPF but they were not stated obviously in the general description. Again this might be due to a lack of documentation.

If we only focus on the plug-in development asset, the three solutions might be equivalent meaning that none of the three solutions have designed the plug-in implementation poorly. However JPF and JSPF do not provide a dynamic framework allowing to quickly find the dependencies between the classes like OSGi does. It will take to the user more time to realize it has an unimplementable solution.

### 2.6.3 Choice of the framework

This was not a complicated choice, OSGi seems a lot better to use.

Not only it has a large community which is able to provide a lot of support, but also there are a lot of books treating the OSGi subject.

Also, OSGi is simple to install: you only have to go to the update site and install the plug-in. There are no dead link or cryptic dependencies : when you want to install OSGi, you install OSGi. It was not the case with JPF which had an OSGi dependency. JSPF is easier to install so if we only focused on quick installation and small development, JSPF might have been a better choice as OSGi is a lot bigger and we wouldn't have used all its features.

OSGi is also more complete than JSPF. If you don't know where you are going, you better take too much than too few. That's why OSGi is preferred to JSPF which might seem easier to use but has few features as well.

Regarding the plug-in development as either choice was the same, this criteria is not very useful for our conclusion. As far as the plug-in management is concerned, OSGi with its service tracker wins a point that the two other frameworks do not have.

## 2.7 Conclusion

In this chapter, we explained the motivation behind the plug-in development for software packages. We defined some criteria on which it mattered to us to define a "good" plug-in

framework. In order to be professional we decided to analyze some plug-in frameworks instead of randomly pick one. We saw several plug-in frameworks that were well known in the computer science field. We presented each of them, OSGi more than JPF or JSPF because there was more documentation.

We saw that OSGi was really complete and useful but that is would not be recommended for small applications to develop quickly without the will to sell them. At the opposite side, we found JSPF which was perfectly suited for small applications which needed to be developed very fast. And the third was JPF which pretended to be like OSGi but as we could not install it, we could not judge of that. Also, as it seemed dead, it would not have been clever to pick this one for a reference implementation of URM (TURM).

Then we had to make a choice so we had to choose criteria that most mattered to us and then see which of the frameworks was the best. OSGi was picked because of its broad documentation, its huge community and the easiness to install. Also as we do not know yet the size of TURM, we will not know if JSPF was more suited for this job as long as TURM does not get a terminal form. But as we did not know where TURM was finally heading, we had to be aware that there might be other users coding after us and that OSGi was the most secure choice to make.

We will thus use a big framework but will probably not use all its features simply because they will not be needed. In the next chapter, before presenting TURM which is a Tool for Usage Rights Management, we will present URM and ODRL which are languages to express rights. As TURM was made to be a reference implementation of URM and as TURM was the software package we focus on in this paper, it would be useful to acknowledge the reader about some Rights expression languages.

# Chapter 3

# Rights Management

One of the main purpose of TURM is to be able to manage license files attached to audio files. Nowadays, the majority of the Belgian population own at least one computer. It is very simple to get songs on Internet by simply doing a Google search. As a lot of people don't understand well how all this work, they don't get the legal issue to this. Even though you can download a song on the Internet does not mean you should. There are laws in every country to prevent people from using one's creation without his agreement. We assume that if people had easy ways to understand how the legal system work with the Internet media they would be eager to behave legally. That's why the usage rights management were made. It is a tool that might help the users to get a better understanding of whether what they have on their computer is legal or not and which rights are granted with the goods they download. It is hard for a lot of people to look down the law to get the specific paragraph related to their situation. There are also some contents on the vendor's website explaining the rights of the users. The main problem with these contents is that they are very long and that as most of the license agreements, most people don't read them and simply click on "I agree". With URM, the users would be provided with a simple way to know what they can do with their media and they will be helped to behave legally.

But in order to manage licenses you must have a rights expression language (REL). The authors of URM didn't reinvent a whole new language but inspired themselves on an other one: the open digital rights language (ODRL). This is a REL that is meant to become a W3C standard. It provides a complete way to express a license though permissions and prohibition. All the licenses belong to one person or one group (party) and he/they can do a restricted amount of actions with the media file they have (e.g. copy, read, ...). This REL also have the concept of Duty that is something you must do in order to get the right to use the license. In general cases it implies paying a certain amount of money or deleting the object after usage.

This section will go in greater details about what is URM and what is ODRL.

## 3.1 URM

### 3.1.1 Overview

URM (Usage Rights Management) was presented by Helge Hundacker,Daniel Pähler and Rüdiger Grimm during the Virtual Goods convention in Nancy during September 2009. Sony BMG and Apple are giving up DRM by means of copy protection leaving no other choices to users than to read the copyright Law or to read the conditions of use in webshops. [HPG09] As a result, a lot of people decided to get files from another source such as peer to peer networks. In Germany, there have been a lot of cease-and-desist letters sent to peer to peer users, often with a monetary compensation demand ( more or less 450 Euros per song).[HLG10]

The idea behind URM is to allow users to manage their music files using the XML-based rights expression language ODRL. ODRL will add a policy to the song which can be read by media library, for example with an extension of Winamp. This REL was not made to force people to behave legally but rather to inform users about their rights about their music files.

### 3.1.2 Usage rights and digital rights

In many countries, a complex system defining the intellectual property exists. A author (a musician) has rights on his creation but can sell exploitation rights to other persons ( music labels) who will get the right to sell the creations of the author. At the end of this chain, the users also have usage rights to tell them what they can and can't do with the goods they buy. Some of them are defined by the vendors and some of them are defined in the country law.

Digital Rights Management is a commonly used expression though it does not have a general and accepted definition. A broad definition could be "Digital Rights Management (DRM) involves the description, identification, trading, protection, monitoring and tracking of all forms of rights usages over both tangible and intangible assets – both in physical and digital form - including management of rights holders relationships".[Ian01]

This definition does not only focus on protection but also on organizational and legal aspects. Vendors own control of their media file forcing their clients to use specific programs and specific players. They often encrypt the content in order to avoid reproduction making this management not very flexible. A song bought on-line from Vendor A will eventually not work with a player that works for the songs bought from Vendor B. But these practices are dying considering that even the market leader Apple decided not to sell anymore protected audio files. DRM might still be useful though because there are other models to be developed for example in the gaming console sector.[HPG09]

### 3.1.3 Usage Rights Management

The main goal of URM is not to replace DRM but to act more like an information policy: it does not enforce people to behave legally but it gives information so they can know whether what they do is legal or not. It was a choice of the authors to keep URM to an information state because a good information policy without an enforcement policy works better than an enforcement policy based on an incomplete information policy.

These licenses can be expressed from 2 writers : the user himself or the authority. When the user writes himself the license, it can be a help for him to organize his media files and see clearly what he can do with them. He will be easily aware when a content expires and when he cannot play it or copy it anymore. When an authority writes the license, it could have a legal value. Indeed, when an on-line vendor writes licenses for his media files, it can somehow work as a guarantee that the user has the legal rights to own the files where in the latter case it could only be used as personal information.

To express the permissions, the authors decided to use the traffic light paradigm.

- green light : you can access the file (that's to say: a license exist and the user can access the file).
- yellow light : no license exists for the file.
- red light : a license exist but the user cannot access the file.
- blue light : the user has the right to forward to a friend

Something we must underline here is that assuming that the user will behave legally, nothing forbid a user to have the rights to share the file without having the rights to play it. For instance if you buy a song you can only listen 10 times and send to 5 people you can simply play it 10 times and only then send it to your 5 friends.

A typical scenario of URM usage might be for someone who rips every CD he buys in order to keep all his music on his computer. He might also begin to download songs via ITunes or other legal on-line music vendors. But after a while he might not now which songs were downloaded and which songs were bought. As there are different permissions because those are too different means of purchase.

Another scenario of URM might be when it is embedded to a peer to peer client. You might be able to see your friends song and which rights they have. Let's say you want the last album of your favorite singer but you don't have money for this. You might wanna check your close friends to see if they did buy the album and if with this album they have the rights to share it with you.

## 3.2 ODRL

The Open Digital Rights Language (ODRL) Initiative is an international effort aimed at developing and promoting an open standard for policy expressions. ODRL provides flexible and inter-operable mechanisms to support transparent and innovative use of digital content in publishing, distribution and consumption of digital media across all sectors and communities. The ODRL Policy model is broad enough to support traditional rights expressions for commercial transaction, open access expressions for publicly distributed content, and privacy expressions for social media. ( ! quote from W3C community group). There has been a core model developed in order to show more precisely how an ODRL policy works. All of the definition belows have been greatly inspired by the Core-Model document from ODRL W3C group.[com12]

### 3.2.1 Technical description

#### Policy

An ODRL policy is centered on the concept of `Policy`. This policy represents a set of `Prohibitions` or `Permissions` but in order to describe what are `Permissions` and `Prohibitions`, we must first describe more basic concepts such as `Asset`, `Party` `Constraint` and `Action`. We will then describe more easily complicated concepts like `Permission`, `Prohibition`, `Duty` and `Policy`.

A typical policy would be "you have the right to play the song you just downloaded but only after fulfilling the duty of paying two euros to the vendor. You cannot distribute this song to your friends and cannot modify its content to create something you claim to be the author of.". Of course a Policy can contain several permissions and prohibitions, but a permission and a prohibitions can only be linked to one policy only. According to the diagram, a policy might not have any permissions or prohibitions but in that case there is no point to do one. This is the same for the Asset part; if there are no assets, there is no use for a policy.

#### Asset

An `Asset` is generally the subject of the policy. An `Asset` is required by `Permission`, `Prohibition` and `Duty` entities because they describe an action that can or can't be performed on a defined subject.

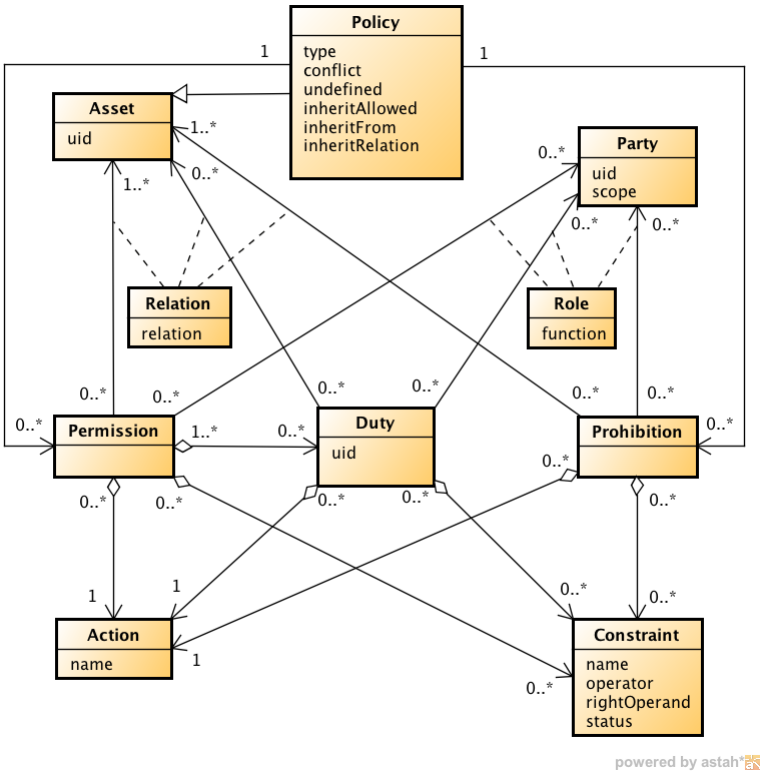


Figure 3.1: ODRL Core model.
[IGPK12]

An asset is generally the good who is the subject of the policy. By good we mean any digital media content: song, videos, e-books, ... The asset can be aggregated. For example you can see a song as an asset, but also a whole CD album.

It has one mandatory attribute:

- uid: the unique identifier of the `Asset`

Since ODRL can deal with any kind of media files, it is not necessary to create more attributes to describe a specific kind of asset. It is though recommended by the ODRL community to use "Dublin Core Meta-data Terms" that are much more appropriate to describe Assets.

### Relation

The `Relation` entity links the `Asset` to `Permission`, `Prohibition` and `Duty` types. The `Relation` entity has an attribute `Relation` that describes the kind of relation the `Asset` has with them. It's default value is "target" which indicates that the `Asset` linked to the other entity (e.g `Permission`) is the primary object to which the permission, the prohibition or the duty applies.

### Party

A `Party` represent a legal person (e.g a physical person, a group or an organization) who will participate to the policy transactions. It is composed of two attributes:

- uid : the identifier of the Party
- scope : the interpretation which should be taken under a specific context (this attribute is optional).

For instance, the scope attribute could have the `individual` value to indicate for instance that the policy applies to one single person. Dually, it can take the `group` value to indicate that the party is a group of more than one persons and that each person has to adhere to the policy. For a more accurate description of a `Party`, it is recommended to use, for instance, IETF vCard meta-data standards that are suited for the `Party` type.

### Role

A `Role` entity for a `Party` is the same kind of relation than the `Relation` entity to an `Asset`. It is used to link a `Party` to a certain `Permission`, `Prohibition` or `Duty` into which the `Party` will be involved. A `Role` has only one attribute called function. It is the function the party takes. It can take only two values : `assigner` and `assignee`. An assigner is the stater of the policy and the assignee is the one that has to respect the policy.

### Action

The `Action` Entity is quite trivial: when related to a `Prohibition` it refers to an action that is permitted to be performed by the assignee. When related to a `Permission`, it refers to an action that is forbidden to be performed by the assignee. For the `Duty`, it refers to the action to be performed.

An `Action` has one parameter, the name of the action that can be performed.

This parameter is supposed to take a list of pre-defined values that still aren't completed by the community group. Even though there are a set of values, there might be some new ones in the future.

### Constraint

The `Constraint` Entity represent the constraint linked to the `Permission`, the `Prohibition` or the `Duty` Entities. Is is a mathematical expression with one operand and two operators. All the `Constraint` Entities must be satisfied to respect the policy. A contraint contains three attributes for the expression of the constraint (name is the left operand, operator the operator and rightOperand is the right operand) and the last one is the current value of the left operand.

For instance, a `Constraint` can be " number of usages must be smaller than 10". In that case, the name is "number of usages" the operator is "smaller than" and the right operand is 10. The optional value status if initialized should be a value between 0 (if the media has never been used) and 10 (if the media has already been used 10 times). The constraint is used by permissions, prohibitions and duties to get a predicate that must be satisfied. They are the way of expressing generally the cost of a song, the amount of times you can play it or share it to your friends.

### Permission

The `Permission` entity grant an `Assignee` to perform a set of `Actions` with an `Asset`. It is composed of the following relations:

- Asset: the `Permission` entity MUST refer to an `Asset` (where at least one, and only one, relation value is target) on which the linked `Action` MAY be performed (this attribute is mandatory)
- Action: the `Permission` entity MUST refer to exactly one `Action` that indicates the granted operation on the target `Asset` (this attribute is also mandatory)
- Party: the `Permission` MUST refer to one or more `Party` entities linked via the `Role` entity (this attribute is optional)

- Constraint: the `Permission` MAY refer to one or more `Constraints` which affect the validity of the `Permission`, e.g. if the `Action` play is only permitted for a certain period of time (this attribute is optional : you can have a song you can do anything with)
- Duty: the `Permission` MAY refer to one or more `Duty` entities that indicate a requirement that MAY be fulfilled in return for receiving the Permission (this attribute is optional as well, there are also really free songs on the Internet that you can read or share with friends)

[IGPK12]

Some example of permissions can be "you are allowed to play the song 10 times" or "you can copy this license and share the song with three of your friends". A permission is at least linked to an asset because you need a subject for this permission. It is also linked to an action you can perform because having the right to do nothing is not very constructive. However, it may neither have a duty (for example if you get the song for free) nor have constraints (e.g. having the right to play a song indefinitely). Having a Party to which the duty implies is also optional.

### Prohibition

Dually, a `Prohibition` entity prevents an `Assignee` to perform a set of `Actions` with an `Asset`. it is composed of the following relations:

- Asset: the `Prohibition` entity MUST refer to an `Asset` (where at least one, and only one, relation value is target) on which the `Action` is prohibited (this attribute is mandatory)
- Action: the `Prohibition` entity MUST refer to exactly one `Action` that is prohibited (this attribute is mandatory as well)
- Party: the `Prohibition` MAY refer to one or more `Party` entities linked via the `Role` entity (this attribute is optional)
- Constraint: the `Prohibition` MAY refer to one or more `Constraint` entities (as for the permissions, this attribute is optional for the same reasons: you can have a song with which you're not supposed to do anything)

A prohibition works the same way as a permission: it defines what you are not allowed to do with a content. Examples of prohibition can be "you cannot use a part of this song to create a new one of which you would claim to be the author of" or "you cannot share this song". The prohibitions' cardinality can be explained the same way than the permissions' one.

### Duty

The `Duty` entity represents an `Action` to be performed in return from having a certain `Permission`. A `Permission` can have more than one `Duty` entities to fulfill and in that case they must be all satisfied in order to get the `Permission`. If there is one `Duty` required by several `Permission` entities it only has to be satisfied once to grant all the `Permissions`. A `Duty` has as attribute the unique identifier that references the `Duty`.

It also has the following relations :

- Action: indicates the operation that MAY be performed (this attribute is mandatory). Note: It is assumed that the assigned `Party` has the appropriate permissions to perform this action.

- Party: a `Duty` MAY refer to `Party` entities with different `Roles`. If no explicit `Party` is linked to as assignee or assigner, the `Parties` with the respective `Roles` are taken from the referring `Permission`. (this attribute is optional)
- Asset: a `Duty` entity MAY refer to an `Asset` (where at least one, and only one, relation value is target) related to fulfilling the `Duty`. For example, a pay `Action` must be linked to a target `Asset` that indicates the amount to pay. The mechanisms to perform this linking/packaging are defined by community Profiles. (this attribute is optional)
- Constraint: a `Duty` MAY link to one or more `Constraints` (this attribute is optional as well)

A duty represent generally what you have to do to get the permission granted. Generally it implies paying the price or accepting to delete the content after the license has expired. A duty's cardinality can be explained the same way as a prohibition.

**Policy**

The `Policy` is the main attribute of the OSGI Policy and is composed of several attributes:

- uid is the unique identifier of the `Policy`
- type is the type of background of the `Permission` (e.g. an offer, a contract, an agreement, a request)
- undefined indicates how to handle undefined `Actions` (OPTIONAL)
- conflict manages the priority of `Permissions` and `Prohibitions`
- inheritAllowed is set to true if the `Policy` can be inherited
- inheritFrom is the identifier from which this Policy inherits from it's parent Policy (OPTIONAL)
- inheritRelation is the identifier for the relationship type of this inheritance structure (OPTIONAL)

There are types described in an other document of the ODRL community group but the purpose of this section isn't to make the reader an ODRL expert. Furthermore there is a lot of vocabulary that should be added to get a full knowledge of the current ODRL. Nevertheless to fully understand a `Policy`, we must explore in greater details the concept of inheritance.

Regarding the conflict attribute, there are only 3 values that can be taken:

- perm means the `Permissions` are more important
- prohibit means the `Prohibitions` are more important
- invalid means the `Policy` is not valid

The undefined attribute must take one of the following values if it is present in the `Policy`

- support means the `Action` is to be supported as part of the policy – and the policy remains valid
- ignore means the `Action` is to be ignored and not part of the policy – and the policy remains valid

- invalid means the `Action` is unknown – and the policy is invalid (default value)

Like the `Asset` entity, some attributes may be added to a `Policy`. The ODRL community recommends to use the Dublin Core Metadata Terms.

**Inheritance in Policy entity**

the inheritAllowed attribute is set as true if the `Policy` can be inherited, it will not be more described in this section.

The inheritFrom attribute in the (child) Policy will uniquely identify (via a UID) the (parent) Policy from which the inheritance will be performed.

The inheritRelation attribute in the (child) Policy will uniquely identify (via a UID) the type of inheritance from the (parent) Policy. For example, this may indicate the business scenario, such as subscription. Such terms MAY be defined in the Common Vocabulary or community Profiles.

There are five properties for an inheritance to be correct:

- Single inheritance is only supported. (One Parent Policy to one or more Child Policy entities. No Child Policy can inherit from two or more Parent Policy entities.)
- Inheritance can be to any depth. (Multiple levels of Children Policy entities.)
- Inheritance cannot be circular.
- The Child Policy MUST override the Parent Policy. i.e.: If the same Action appears in the Parent, then it is replaced by the Child version, otherwise the Parent Actions are added to the Child's Actions.
- No state information is transferred from the policy in the Parent Policy to the Child Policy

### 3.2.2 A small example

This example explains how to create a simple offer policy. There is in this case the party http://example.com/sony:10 who wants to offer a music file (http//example.com/music:4545). We can see that he is the assigner and not the assignee and that the music file is the target of the permissions. We can see that there are two permissions with this music file: the right to play and the right to copy. The right to play is unconditionnal (you can play it any times you want) but implies fulfilling the duty of paying http://example.com/ubl:AUD0,50. For the right to copy, it is quite different because you have a constraint defining the number of copies you can make. In this example, this count is set to 1. So in shorter terms, this policy allows to play a song and to copy it once if we pay the price of 0,50 AUD to Sony.

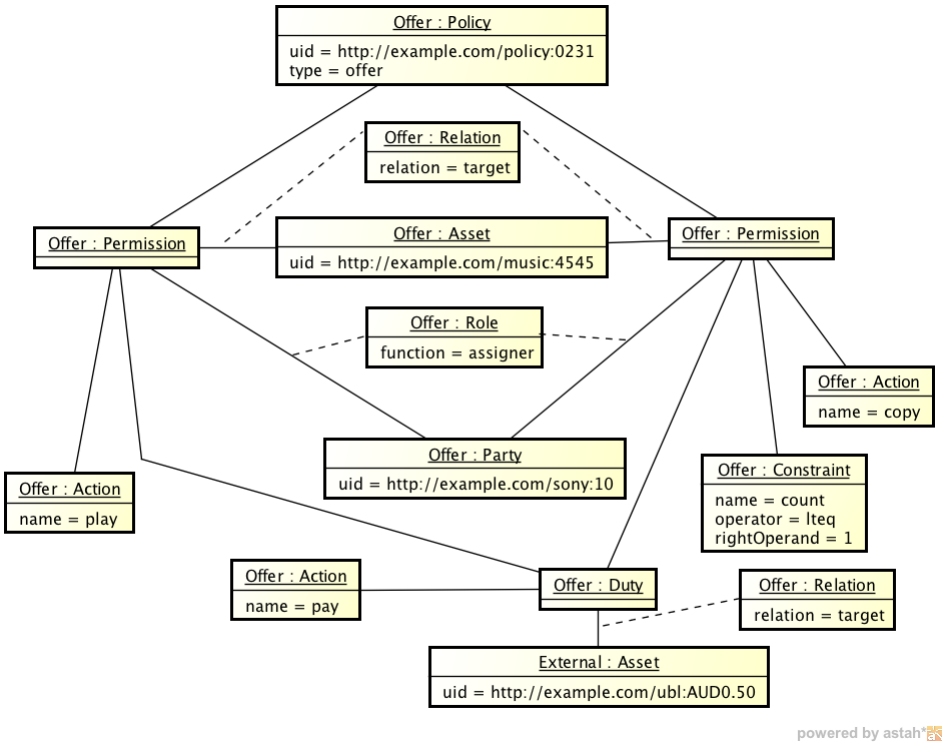


Figure 3.2: An instance of an offer policy
[IGPK12]

### 3.2.3 Conclusion

In this chapter we have defined what is a rights expression language and what were URM and ODRL. We got a more technical description in order to show that this isn't simply an idea but a really well thought concept (ODRL is in its way to become a W3C standard). This part was important to validate the utility of TURM and the purpose of its creation. Moreover as the description of TURM implies a general presentation, this chapter gave the basis to understand what the

# Chapter 4

# TURM

TURM (Tools for Usage Rights Management) is an application designed to be a reference implementation for URM (Usage Rights Management). At first it was the idea of Daniel Pähler who wanted to get a small Linux implementation of a license manager. Even though at that time it could only display the rights for a song, it became a broader application. Options were added to manage the licenses more efficiently and export them. There is also a peer to peer interface that was designed to fulfill the ideas of legal sharable content.

As back then it was a small program, it was implemented using a classic monolithic way. But as it spread, it became more and more complex, becoming a huge monolithic bloc which wasn't what its creator wanted at the first time. Plus, he did loose a bit of control of TURM giving thesis about a feature to implement in TURM and so the application hasn't been tested completely for a while.

So before presenting the plug-in implementation of TURM, we must first get ourselves familiar with it and get an overview of it, of what could possibly be done to improve the modularity and why some features which appeared to be a good idea to embed into a plug-in were dropped.

The next chapters will give a quick overview of TURM, followed by a more detailed description in order to understand how TURM was programmed and then we will give an explanation of the choices we made to create plug-ins for TURM and why we implemented them that way.

This chapter will give an overview of TURM's features, describing how it works. We will explain each feature but won't go into implementation detail to get a rather abstract view of it.

For this purpose, we bought some music files and created user licenses with them. This will give a better view of how everything work much more easily than by explaining everyting without showing any screens of what the features and the menu's are like.

This chapter will be cut into pieces. The first piece will explain the media and license features which regroup everything a user can do with a license ( exporting license, getting more information about licenses) and what a user can get from a media (information interface, media organization and management, meta-data information). The second piece will be a more "meta" part because it will present the configuration features (language, destination files). As the CUP part isn't fully integrated to TURM yet, it will not be discussed in this document.

## 4.1 TURM media and license features

This is TURM's GUI main page. It shows the audio files that have been added to TURM. The data showed are the path of the file, the data type, the Rights and a hash-value. The two first column don't really need explanations but when we look at the third, we see

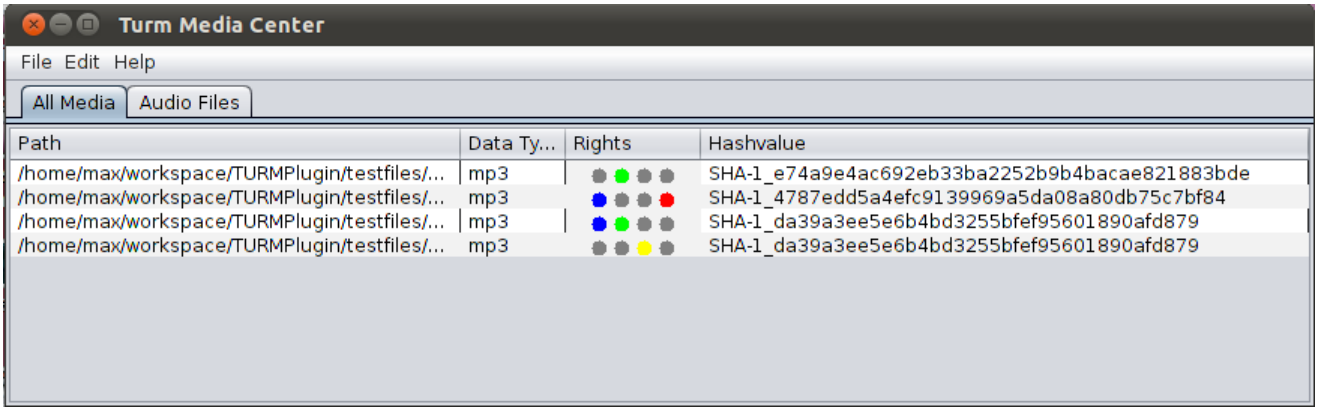


Figure 4.1: TURM main screen.

different colored circles. On the first file, only the green light is on meaning that we have the rights to play it but not to share it. On the second file, only the blue and red lights are on; That means that we can share this file but that we aren't allowed to view its content. This could mean that we had the right to play the song 5 times and share it to 2 friends and that we already have played it 5 times. For the third file, we have the rights to play it and share it and for the last file, the yellow light means we have no licenses related to this file.

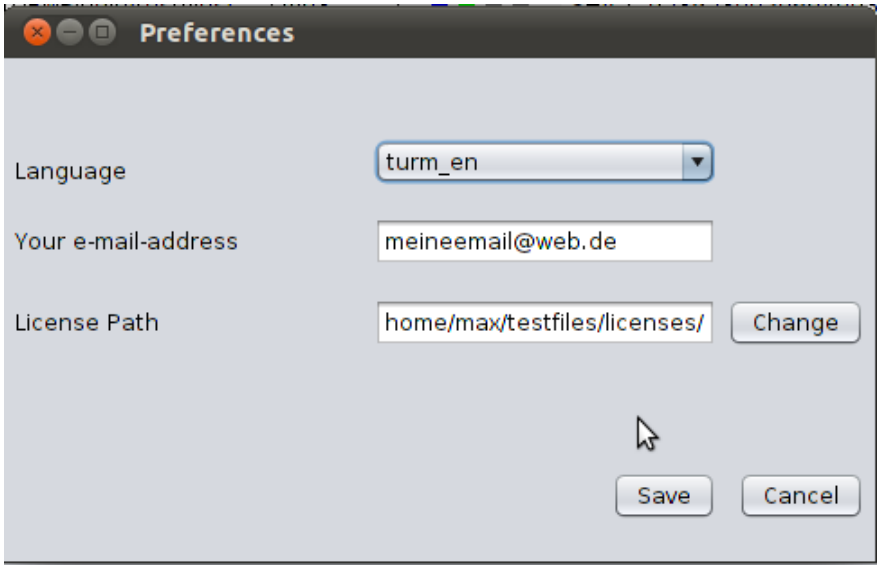


Figure 4.2: TURM settings menu.

In the file menu are located all the functions related to file loading. There we can load a file or an entire folder. There is also a menu to change the language because TURM's default language is German. However the authors created an English option. Actually those are the only two languages of TURM. In that settings page, the user can also fill in his e-mail address which is used as an identifier for the licenses. There is also a license path in order to stock all the licenses at the same place.

The second tab of the main screen contains more accurate data about the files. In addition to the hash-value, the rights column, the data type and the path, there are information about the song itself such as the artist, the title, the album and so on. There are also comments about how the file was found. While right clicking on a file, the user can access to options regarding the license files.

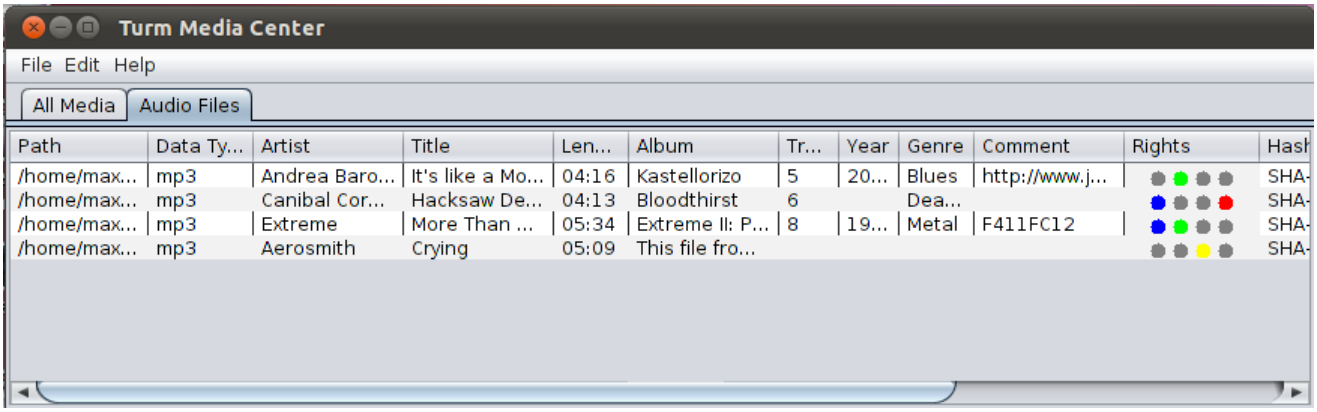


Figure 4.3: TURM main audio screen.

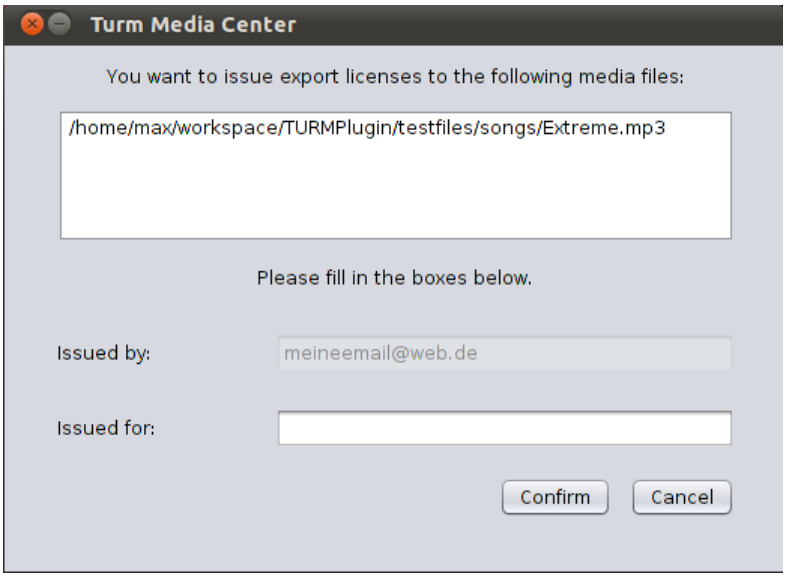


Figure 4.4: An export license menu

To create an export license, only an identifier of the person who is the target of the license is needed. As the e-mail addresses are regarded as an identifier, it is enough to create a new export license. However if the assigner has no right to export the license, an error will occur.

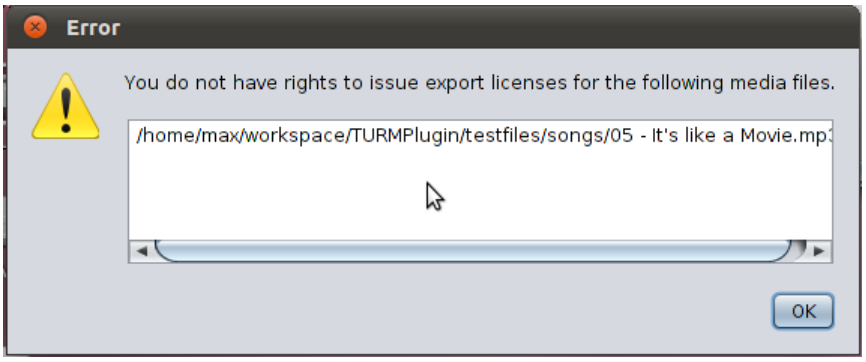


Figure 4.5: An error that occurs if an export license is requested for an unlicensed file

There is also a tab with more details.

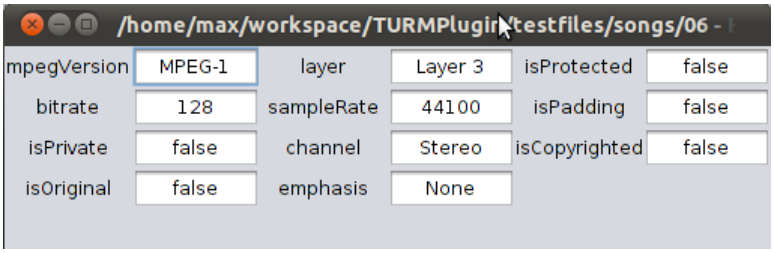


Figure 4.6: More Meta Data

### 4.1.1 Future work

There is a feature currently being implemented on TURM. This feature will allow TURM to work with media players and peer to peer client which was the main objective behind TURM.

The university of Koblenz has also developed a format in order to increase user awareness about his permissions and prohibitions as well as his duties. this format is called Formosa. This format also allows users to fix themselves limits. This limit is called a burden. For example if an actor has a wallet of 100 euros and if the amount of all the duties he has been engaged for are above this limit, the actor will be warned that if he had to pay all his duties at once, this amount would exceed the limit of his wallet. It also checks whether the action he wants to do is legal or not. If those "bad scenario" occur, the action is deemed insecure.

As TURM gives the user an intuitive feeling about his rights, it might be fit to the implementation of Formosa. So far no further technical choices have been made.

## 4.2 TURM configuration features

This section will be split into pieces. First we will make a more technical description of TURM allowing the readers to understand more how the features work and how they are assembled to work with each others.

Then we will make a section explaining the cleaning we did to the code. Besides a plug-in implementation, we also cleaned the code in order to get a full working version of TURM because as mentioned above, nobody had tried all the pieces together for a while.

### 4.2.1 Overall description

The starting package is the GUI. It is linked to the facades and the events_and_listeners packages. The facades package is just some default graphical settings to create the user interface. The package events and listeners regroup all the listeners for the GUI. Then there is a link to the core because the GUI instantiates TURM's core.

The core then instantiates the two TurmFeatureManager. One will regroup the hash calculators regrouped in the package hashCalculator and the second one will regroup the payload extractors regrouped in the package payloadExtractor. Those two packages were originally in the core, but we decided to extract them in order to keep most of the classes away from the core.

The hashCalculator package regroups the functions related to the hash values calculations. It provides the licenses with an unique identifier for each music files. It is requested by the licenses because they need a more precise identification than just the title of the song. Actually the only hash calculator supported is SHA-1. Back then it seemed a good idea to the authors to implement this as if there would be other standards coming and as

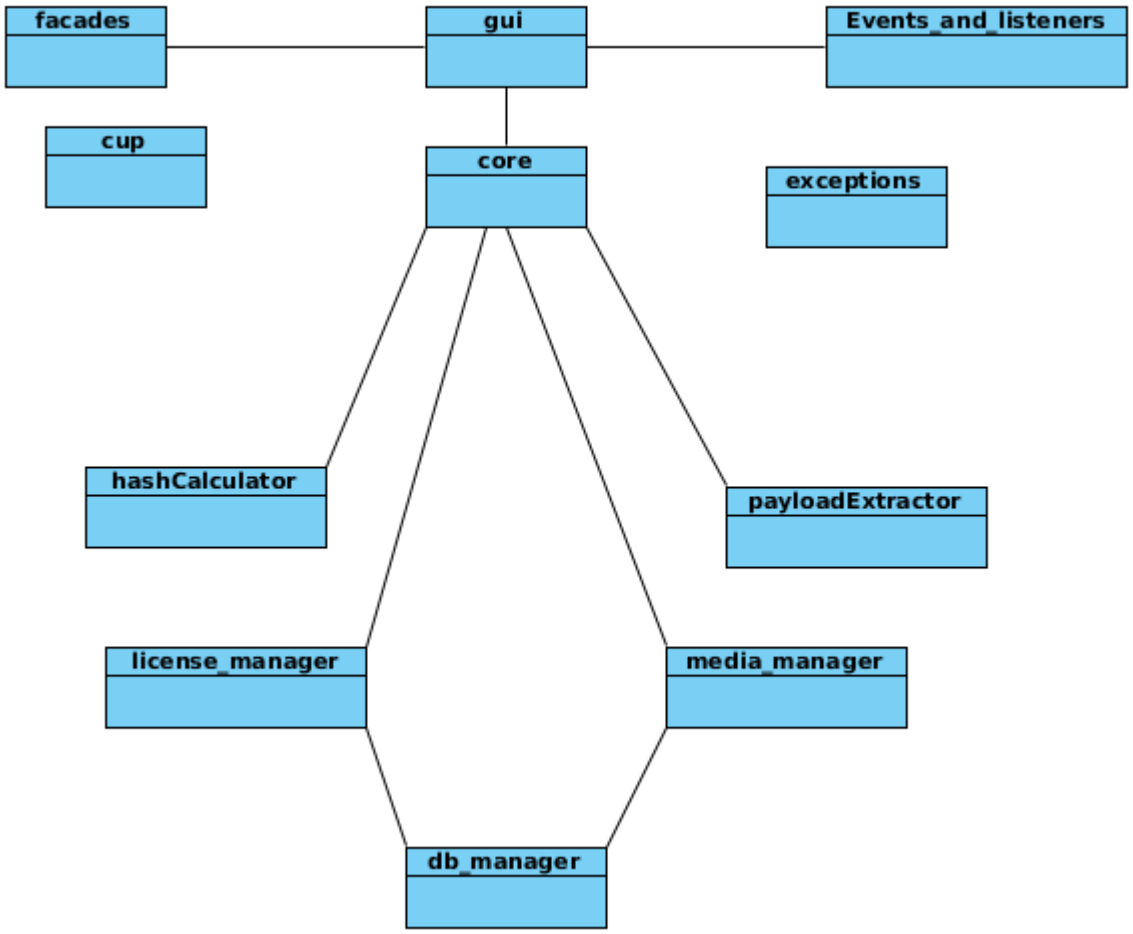


Figure 4.7: A diagram showing TURM package dependencies

if they would have to keep up do date. After some discussions with Daniel Pähler, it seems now a bit irrelevant to make this kind of structure because SHA-1 will probably be kept.

The payloadExtractor regroups all the functions related to data extraction from songs. There are two kinds of data extraction: the data about the song (artist, track number, album, ...), and the payload. The payload here represents the musical content of the file. So it is basically the file but without all the headers and meta data about the song. It is used to create the hash value created by the hash calculator. Actually, the only supported format is MP3. This was though a good idea to keep this implementation in a "pluggable way". The media extensions are improving much faster than hash algorithms. Moreover, only one hash algorithm is needed to create a hash-value by opposition to song reader which would optimally need to read all possible extensions in order to be exhaustive. The core then also initializes the license manager and the media manager which will load the licenses and the medias from the database. All interactions with the database are made through the license_manager and the media_manager packages.

The license_manager package contains license reader and writers. The two supported license formats are ODRL and URM. It has been implemented that way because one of TURM's goal is to be a reference implementation for URM and ODRL. There is thus no need of other kind of licenses in the program. Along with the license reader are class that implements URM and ODRL licenses. For the writers, Turm only supports XML writing right now but as the URM and ODRL licenses are represented that way, there is no need neither for other kind of exportations.

The exceptions package is not linked in the diagram, not that it isn't used by TURM, but as it is a package regrouping all the custom exceptions made by the authors, this package is linked to most of the other packages and it would have ruined the diagram to add links from the exceptions packages to all the other packages.

The cup packages is not linked because it has been implemented, but not yet integrated to TURM. We won't describe it in greater details, we will just say that this package contains a first draw of a peer to peer implementation for TURM who will allow it to work as a normal peer to peer client with the license management feature. That means that when a user looks for a song with this peer to peer client, he will have songs but also licenses that are related to them. HE will thus see whether he can download and own the song or not.

## 4.3 Core

This package is the main package of Turm. As there are neither many classes nor many operations, it is possible to put everything in one diagram and still stay clear to the reader.

### 4.3.1 Turm class

The main class is the core. It contains all the managers in one class and is more a class meant to instantiate and link than a class who will perform a lot of business logic operations.

Concerning the attributes, they are quite trivial. The first is an instance of TURM in order to only get one running instance at a time. The pem is the PayloadExtractorManager. It contains the list of all the extensions supported by TURM from which the payload can be extracted. the hcm is the HashCalculatorManager. It contains the list of all hash value calculators implemented in Turm. There were supposed to be many of them but the author realized it would be more efficient to keep SHA-1. The cm is the only value on which an access is granted, it is the configuration manager. There was a ConfigurationManagerOld class in the program which has been deprecated in order to use the ConfigurationManager class as an external library.

The mm is the Media manager, it manages the media, interacts with the GUI to update information about media and store every change in the database. It can also access to meta data structure from extensions (currently MP3 is the only fully implemented meta data structure). The mlm is the class responsible for the mapping between licenses and media. It will browse the license directory and try to match every possessed licenses with a media stored in the database. The lm value is the license manager. It has the same function as the media manager but for the licenses. It is in the license_manager package that the licenses structure are stored. Actually only ODRL and URM licenses are implemented but as the goal of TURM is to become a reference implementation for those licenses, there is no need to implement more of them.

The ff value is the FacadeFactory class responsible for all the GUI structures. It is in the core because as the core initialize all the important classes, it also has to configure the GUI.

Finally, the dbm is the database manager. It has been implemented using hibernate but as some classes are undocumented, it was hard to find what they were there for. Additionally, as hibernate is an external library, it is hard to find all the documentation when a database issue occurs.

Regarding the operations, there are mostly getters in the Turm class. Except from the constructor which is private due to the last operation : getInstance(). The initializeConfigurationManager() operation is meant to instantiate all the default values of Turm. It is composed mostly of paths: the license files path, the media files path, the database path. There are also constants about language files and the folder where the properties file containing additional constants is located.

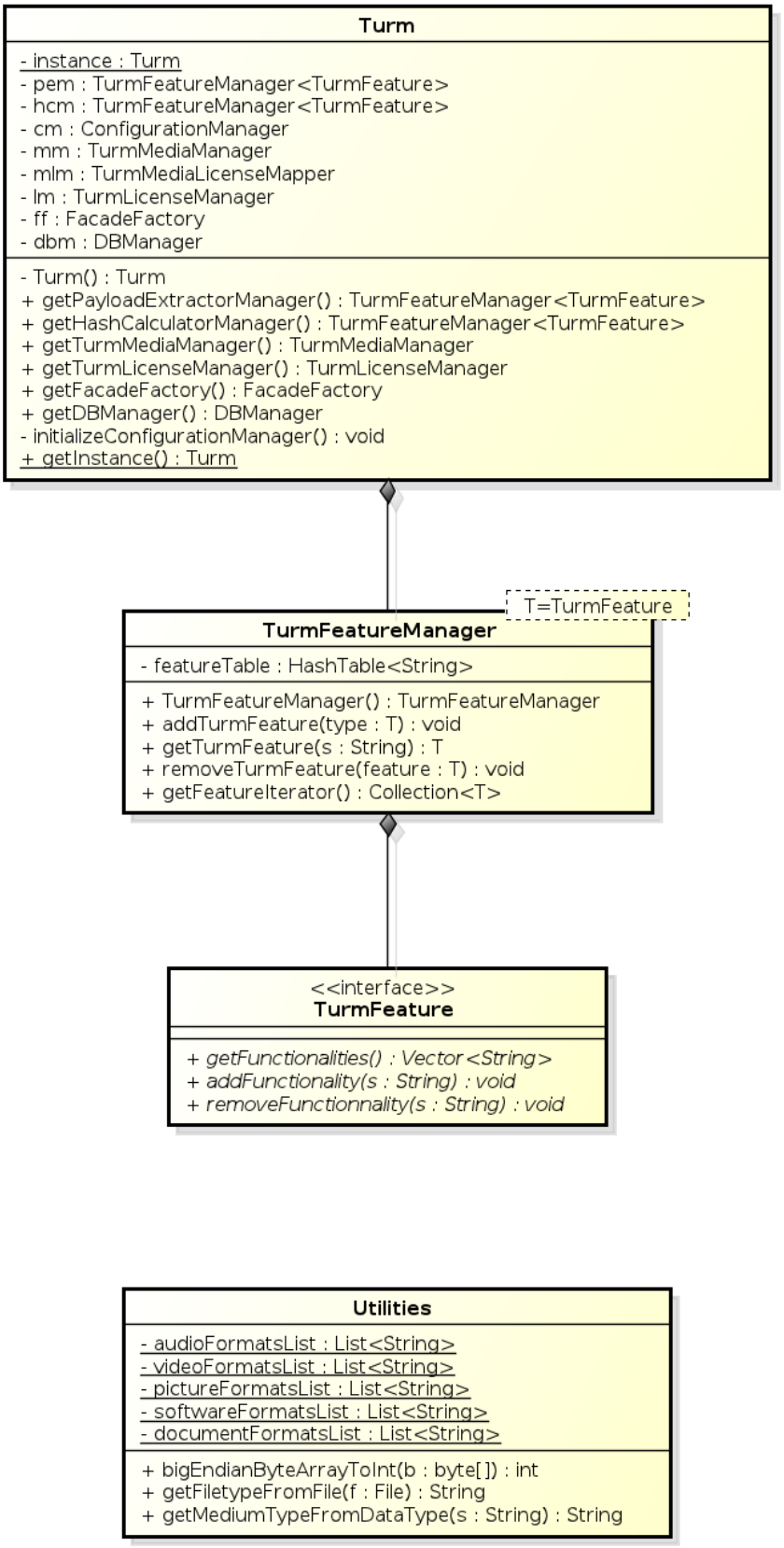


Figure 4.8: a part of TURM's class diagram related to the core package

### 4.3.2 TurmFeatureManager class

The TurmFeatureManager class is the most generic class of TURM. Is is a manager for any classes that implements the TurmFeature interface. It was designed that way in order

to only add a TurmFeatureManager if new features were added. Let's say a user wants to add a converter in Turm in order to convert MPEG files into AVI files, he would just have to add a new TurmFeatureManager who would manage MPEGAVIConverter class which would be an implementation of TurmFeature. It is more complicated than that but this is the general idea behind the TurmFeatureManager's creation.

It has a featureTable which is a hashTable defined on Strings. This hashTable is a pair of key and values. Each key refer to an identifier for the TurmFeature and the value is the class itself. For example, the PayloadExtractorMP3 has the values <"mp3",instance of PayloadExtractorMP3>.

The constructor simply creates a new empty hashtable. The add and remove feature operations simply browse the hash-table to remove the element or to see if there are no elements of the same key/value added. To get a specific TurmFeature, only the key which is the identifier for it are requested. There is also an iterator in order to get all the TurmFeatures. This is used during the browsing of the hash-table.

### 4.3.3 TurmFeature class

A TurmFeature as its name says is a feature of TURM. The currently implemented features are hashCalculator and PayloadExtractor. To understand how this structure work, we'll take the example of the payloadExtractor. a feature would be to get a payload extractor for MP3 files. In order to do this, the getfunctionalities() operation should return a vector containing only the MP3 extension to specify that the payload extractor only work for MP3 files. To add a functionality, the extension on which the new payload extractor should work must be defined. For example, to create a payloadExtractor for MPEG and MKV, the vector must contain "MPEG" and "MKV". To remove a functionality, the user has to input the functionality that the payload is able to manage and that he wants to remove.

## 4.4 GUI description

The GUI is coded using the MVC pattern, meaning that the whole graphical interface is separated in three packages, the controllers, the views and the model. On this sequence diagram is the initialization phase which has been cleaned from inter methods in order to not confuse the reader.

The main class first instantiates TURM which is the main class then uses it as an input for the GUI. To instantiate it, we use a singleton pattern in order to not use more than one instance of TURM which was a bug of the system when we started working on it. The TURM class centralizes all the features. TURM first initializes the main classes of other packages that's to say it creates a new TurmFeatureManager for the HashCalculator classes and another for the PayloadExtractor classes. It then configures the different paths: the path for the license files, the path for the media files and the path for the database. The other global constants initialized are the language, the logo and the user e-mail.

With the new TURM, the FacadeFactory class instantiates the facades for the media manager, the license manager and the configuration manager. When everything is instantiated, the MainWindowController links the Facades with the listeners and prints the window.

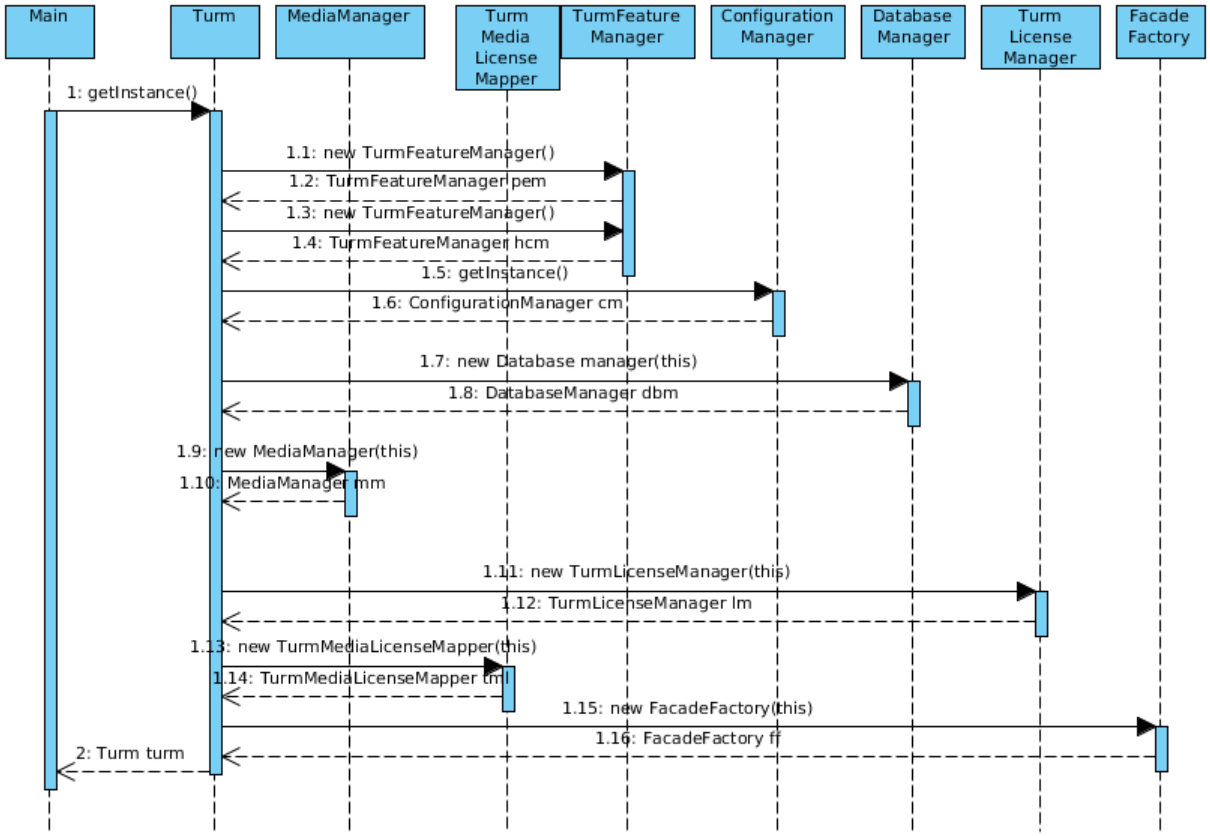


Figure 4.9: Turm instanciation sequence diagram

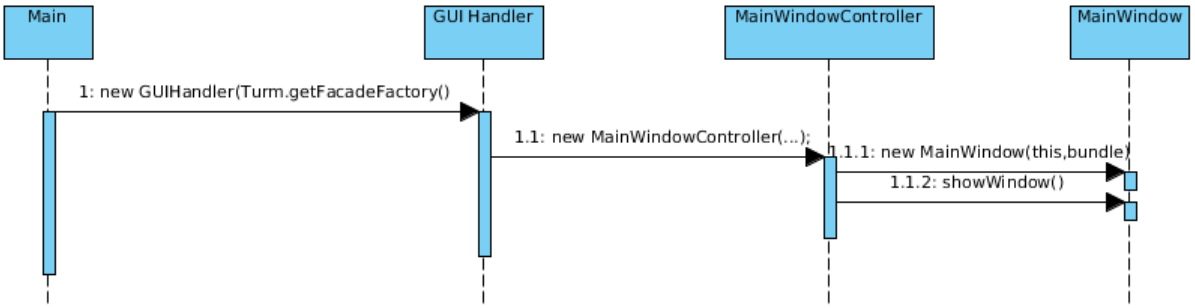


Figure 4.10: GUIHandler instanciation sequence diagram

## 4.5 PayloadExtractor package

This subsection will describe in greater details the payloadExtractor package. This package has been designed to regroup all classes related to payload extraction or more generally, payload management. Initially, this package was located in the core but as the specification of the code aimed at cleaning the core from the most things possible, we decided to move it to it's own package. If TURM's architecture is meant to be plug-in like, this package's size won't increase: all the further supported formats won't be in the package but in a different bundle. If on the other hand, TURM's architecture is meant to remain monolithic, then making the payloadExtractor package was a good choice otherwise TURM core's size will greatly increase with classes that don't really belong to the core. We will in this section go in greater details in the functionalities that the payloadExtractor package can provide.

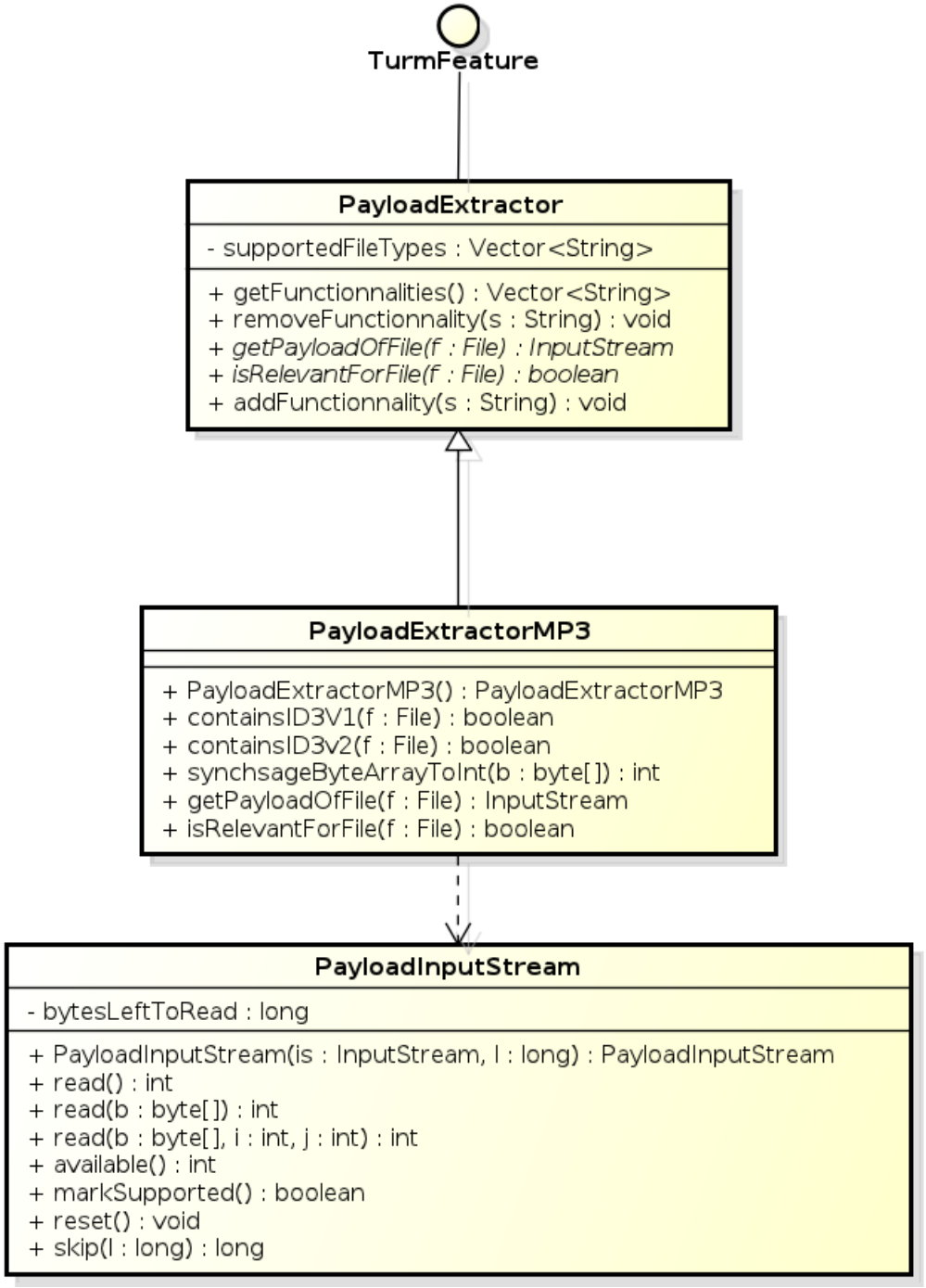


Figure 4.11: a part of TURM's class diagram related to the payloadExtractor package

### 4.5.1 PayloadExtractor class

We will begin with the PayloadExtractor class which is the main class of this package. the PayloadExtractor class implements the TurmFeature interface. It has a private attribute : supportedFileTypes. It is a Vector<String> . This attribute has a getter (getfunctionalities()) which returns also a Vector. Operations are provided to remove or add functionalities. That's to say, if TURM already has a payloadExtractor for MP3 and that the user wants to use his own, he can remove the MP3 functionality from the TURMFeatureManager. Those are rare case, but the operations are there anyway in order to be prepared for such scenarii. The last two operations are abstract and will be described in

more details during the explanation of the PayloadExtractorMP3 class. Those operations are getPayloadOfFile(File f) and isRelevantForFile(File f).

The PayloadExtractor class is the general class for all the extension support. For example, to create a PayloadExtractor for the MP3 format, this class must be a subtype of the PayloadExtractor class. The only supported extension here is MP3 but the diagram won't change for other extensions who might be coming. The PayloadExtractorMP3 class posess the operations needed to extract the contents in an MP3 file.

It has to attributes which are deprecated so we won't discuss them. Its constructor just creates a PayloadExtractor and add in the supportedFileTypes vector the String "mp3". It also had an operation to get the file's payload. Without going into long details, this operation just take the file and try to remove the flags from it. For example the class has 2 methods to check for flags : containsID3v1 and containsID3v2. If the file doesn't contain these flags, no bytes are removed from the file. There is also an int called the synchsafe int. A synchsafe int is an int coded with bytes whose most significant bit is 0. That leaves only 7 bits of information for each bytes. It is used in ID3 tags for MP3 format.

Finally, it has to deprecated operations : synchsageintToInt( int i) and getID3Versions(File f). We did not discuss the return type of the payload extraction operation. We will details it now because it is an implemented type which is a subtype of the FilterInputStream class.

It contains an attribute which give the class information about the bytes left to read. In order to be nstantiated, a PayloadInputStream will call its super type constructor and set his bytes left to read attribute to the attribute passed to the constructor.

The read() operation is the same as the one of its super-type except from the fact that the PayloadInputStream doesn't read when its attribute value is 0.

The read(Byte[] b) operation re implements the super type's version because not all InputStreams relies on the above operation ( read()). To be performed, it calls the operation read(Byte[] b, int i, int j) which is also reimplemented for the same reasons. It works the same way as the read() operation in the sense that it follows the super-type's logic but takes into account its attribute. It is the same for the skip(int n) operation.

normally the bytes left to read should always be smaller or equals to the number of bytes, but the author didn't want to take a risk and reimplemented the operation anyway. For more simplicity, the markSupported() operation will always return false. It does not support marking or resetting as well so the reset() method only throws an exception.

## 4.6 HashCalculator package

This section will present the hashCalculator package. This package's logic is the same as the payloadExtractor package because they both are implementations of the same interface: TurmFeature.

### 4.6.1 HashCalculator class

The HashCalculator class is the main class of this package. Just as the PayloadExtractor class, it provides a generic structure for the hash calculators that will be implemented for TURM. It has a single attribute which is the list of the supported algorithms. There should not be any problems to create a hashCalculator for 2 algorithms but it still would be cleaner to only support one algorithm.

The main functions inherited from the interface are getfunctionalities(), addfunctionality(String s) and removefunctionality(String s). those function as for the payloadExtractor class allow users to add or remove supported algorithm from a certain hash calculator. For example let's say a HashCalculatorSHA1MD5 is already installed but that the user wants to use his own version of a MD5 hash calculator, then he can remove the MD5 functionality from his HashCalculatorSHA1MD5 and use his MD5 hash calculator.

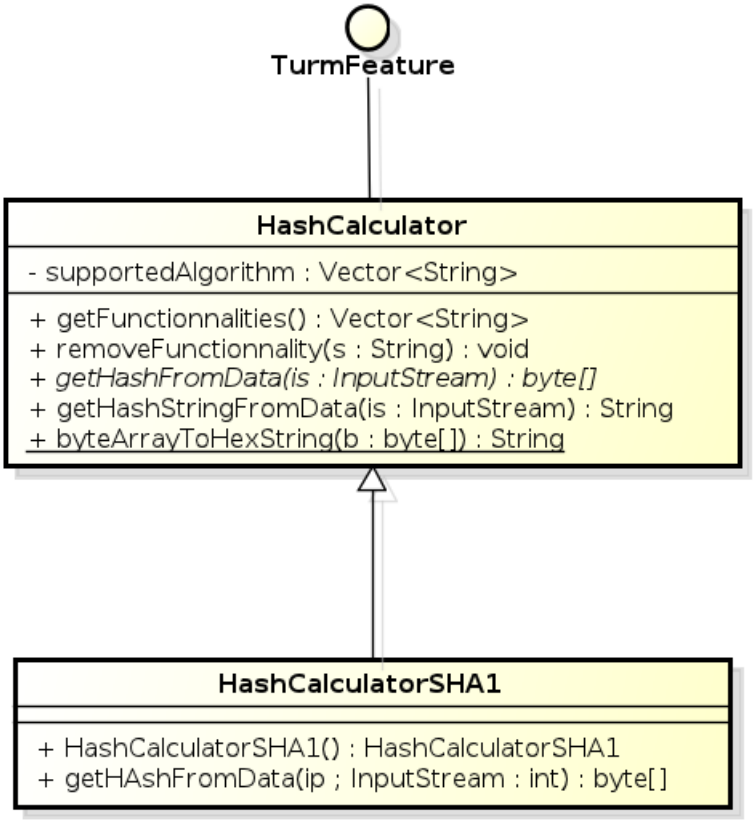


Figure 4.12: a part of TURM's class diagram related to the payloadExtractor package

The specific functions of the hashCalculator package are getHashFromData(InputStream is) and getHashStringFromData(InputStream is). Those functions are made to calculate the hash from a specific input. The getHashStringFromData(InputStream is) is implemented in the abstract class because it is just a call to getHashFromData(InputStream is) and as this operation returns a byte array, the getHashStringFromData(InputStream is) calls a converter and returns the result.

### 4.6.2 HashCalculatorSHA1 class

The hashCalculatorSHA1 class is a subclass of the HashCalculator class. As its name states, it provides a hash calculator following the SHA1 algorithm. There is not much to say about this class, except that it encapsulates the JRE implementation of the SHA1 algorithm. It's only operation besides its constructor is the implementation of the abstract getHashFromData(InputStream is) of the super class. Its constructor calls the super class's one and add SHA1 to supported algorithm. By opposition to the payloadExtractor package, the supported algorithm of the HashCalculator classes must not be set with lower cases only which is the case for the PayloadExtractor classes.

## 4.7 Exceptions

The exceptions implemented in TURM do not really need a lot of explanation. They are all a subtype of the TURMException class. The authors divided them in 3 kind of exceptions. The first is the exception regarding the GUI thus which occurs when there is a graphical problem. The second kind is the plug-in exceptions. It managed errors who occurs with plug-ins. As there were not yet plug-ins implemented, this exception is never thrown. The last kind is the configuration errors. As TURM has to deal with folders of licenses and songs, and as it has a lot of constant values located in a configuration file, those exception seemed relevant. The problem is that the ConfigurationManager class is represented as a library and not a class. We can thus not access to the source code.

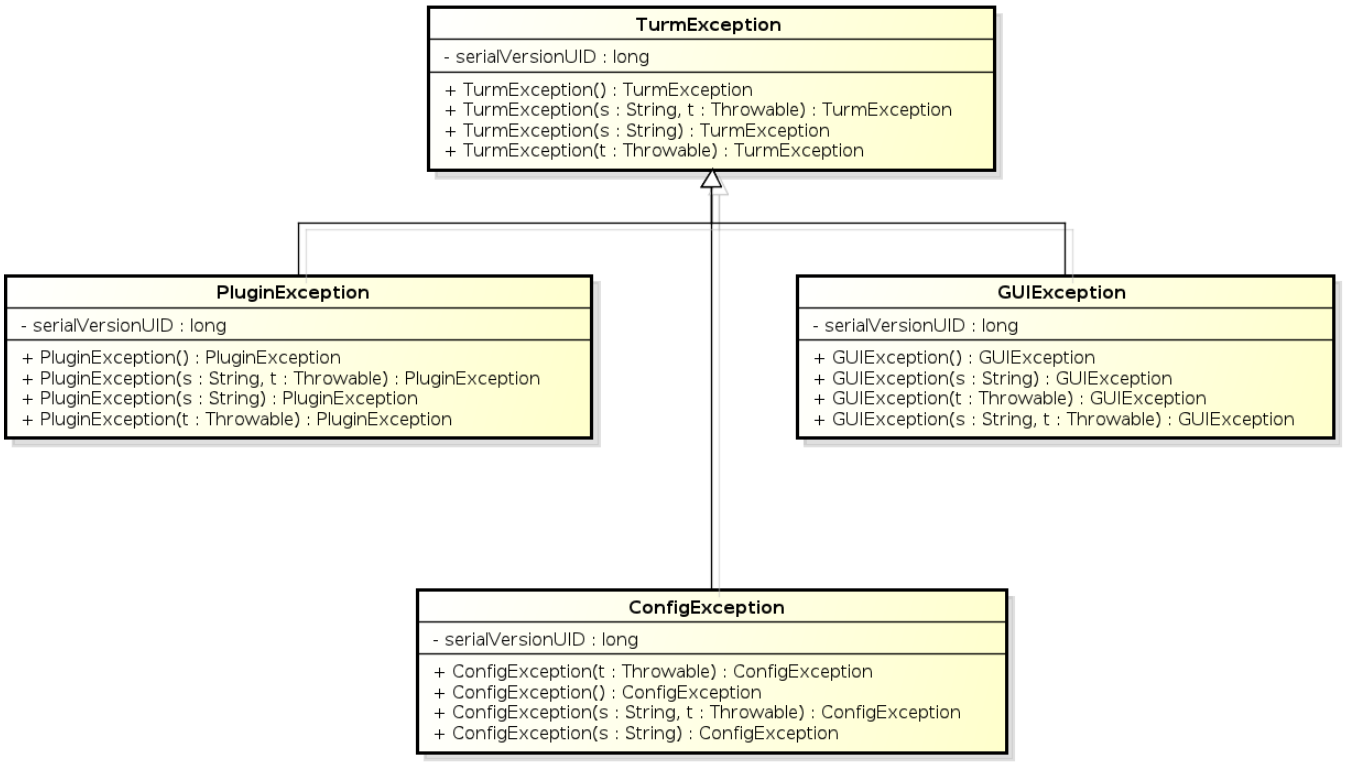


Figure 4.13: a part of TURM's class diagram related to the exception package

Each Exception has been implemented exactly the same way: There are 4 constructors. The first is the default constructor with no parameters. The second includes a message and a throwable. The third and the fourth are with only a message or a throwable.

## 4.8 Media_manager package

To understand better the package media manager, we had to cut it into pieces. Some classes are just data storage classes and they have thus a lot of attributes and their getters/setters. The diagram **??** shows only the classes and their relations without discussing their functions. Every class will be detailed and if there is a non trivial one, there will be a graphical representation of it. As we can see in the package media_manager there is a sub-package media whose utility is only to store and represent information about media. Those classes are observable and do not contain "business code".

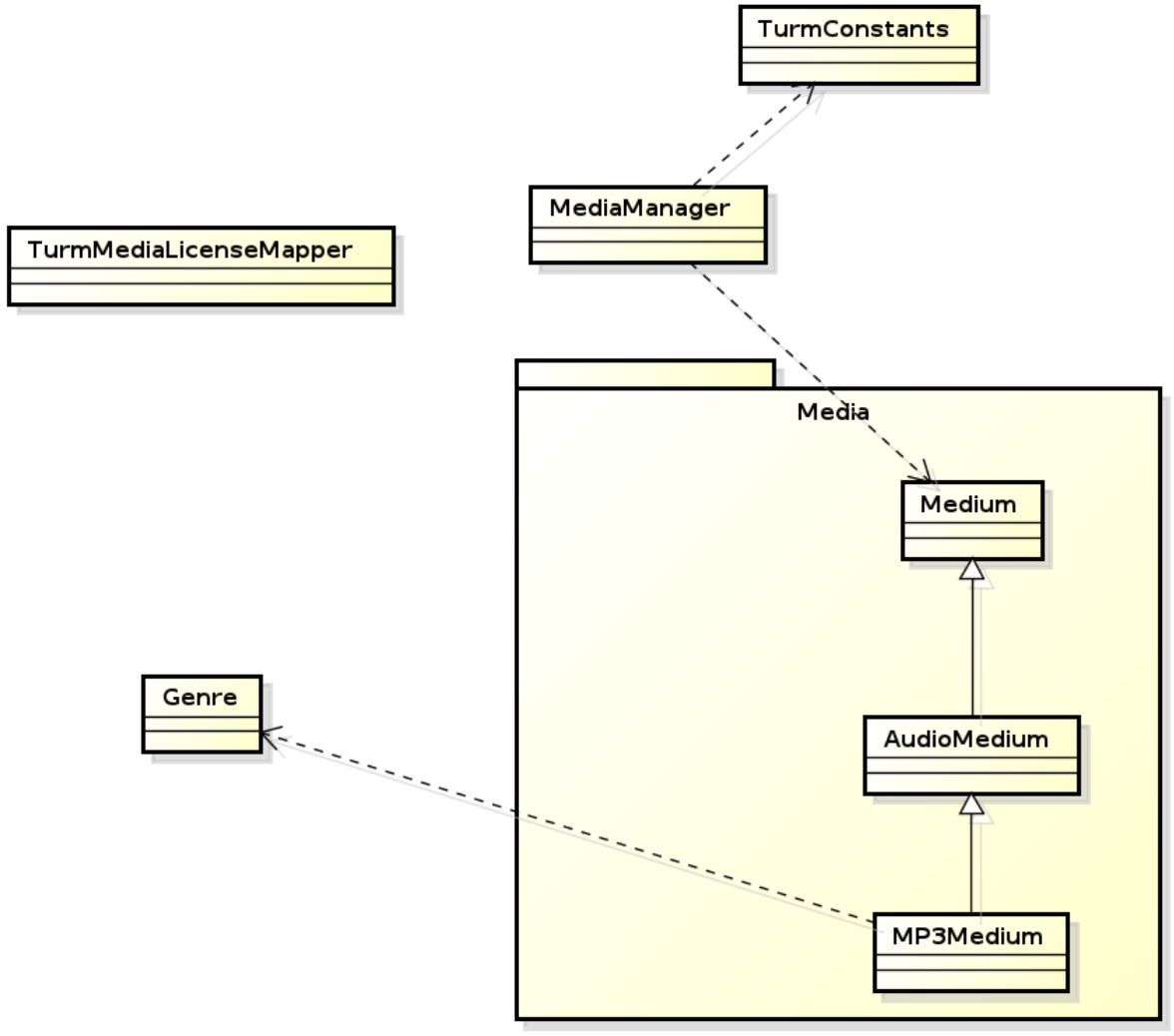


Figure 4.14: a part of TURM's class diagram related to the media manager

### 4.8.1 TurmConstants class

This class contains all the constants needed for TURM. As there are no operations and only attributes it was not usefull to make a diagram out of it. As every name is trivial, it would be less enjoyable for the reader to have the name of the constants in the text.

They are separated into sections though: the constants for the ConfigurationManager, the constants for the licenseManager and the licenses, the constants for hash algorithms, the constants for media types and the values of the names of the setters of the Medium classes.

The first group contains the user e-mail, the licenses files path, the filterList the database path and the file path for the songs. The last one may not be usefull anymore, but as we do not have access to the core we cannot tell if the last one is used.

The second group contains the path of the ODRL right to give, the path of the ODRL right to play and the location prefix of the files (in this instance, the files are located on the user's hard drive so the prefix is "file://".

The third group contains constants about the hash algorithms: the name of the SHA-1 algorithm and the prefix used for the hashCalculations (all the hash values calculated with the SHA-1 algorithm will be prefixed by "SHA-1_". It might not be a good way to implement it though because if we plan to turn the hash calculators into plug-ins, there shouldn't be constants regarding hash algorithm in the core or the class would have to be modified every time an algorithm will be added which is what we are trying to avoid. The group also contains the location of the Medium package in the program and has constants referencing the Medium class and the operation readMetaDataFromFile. Those are used in the media manager during the search for classes that match the Medium class (e.g. Medium and its subtypes).

The fourth group contains miscellaneous constants about the medium: allmedium, audiomedium, videomedium, documentmedium, picturemedium, softwaremedium. Those are used when a media is created in TURM. When a new media is added, it must be of a certain type. Those constants make sure all the media have the same meta tag. For example to state that a file is an audio file a tag "audio", "music", and so on can be added. This constant will make sure every audio file has the same meta tag.

### 4.8.2 Genre class

This class doesn't deserve diagram or long explanations. It was designed to overcome a bug of the jaudiotagger library. Sometimes when the operation to get the genre of the file returns an integer. This class just matches the integers with the genres of the library.

### 4.8.3 TurmMediaLicenseMapper class

This class links together media files with their licenses. It is a class which has relations with GUI as it implements Observer, MediaManagerEventListener and LicenseManagerEventListener classes.

The attributes are quite easy to get. There is the core class : Turm. There is also a list of media files which are recognized by TURM. The third and fourth attributes are list of licenses: one is for the licenses which are not already mapped and the second is for the licenses which are already mapped to a media File. Finally there is the user's email address.

The constructor only take the main class as parameter : Turm. To begin with, it instantiates its own Turm attribute, then adds in TurmMediaManager and in TurmLicenseManager EventListener's with itself as attribute. In order to get notified if the e-mail address of the user is changed, it adds an observer to the configurationManager. The mediaList is instantiated by getting the MediaList from the TurmMediaManager and the unmapped licenses are created the same way, by getting the LicenseList from the TurmLicenseManager. The constructor instantiates the e-mail address of the user and then calls the initialMapping operation.

#### Internal operations

The initialMapping goes through the music folder and try to match each licenses with a music file. There is a DLocation which gives the location of the media file for a license on the owner's computer. The first thing to do is to identify the user and check if he is the assignee of the license. Then we must check the DLocation ( location where the mapped file is supposed to be) if there is a file we then check that its hash-value matches the license's hash-value and then we make the mapping (calling the operation setLicense for the Medium class and put the license in the "already mapped" list instead of the "free to map list". If no file is found at the DLocation, we look if a file of the user's folder match the license's hash-value. We also then update the Dlocation of the file.

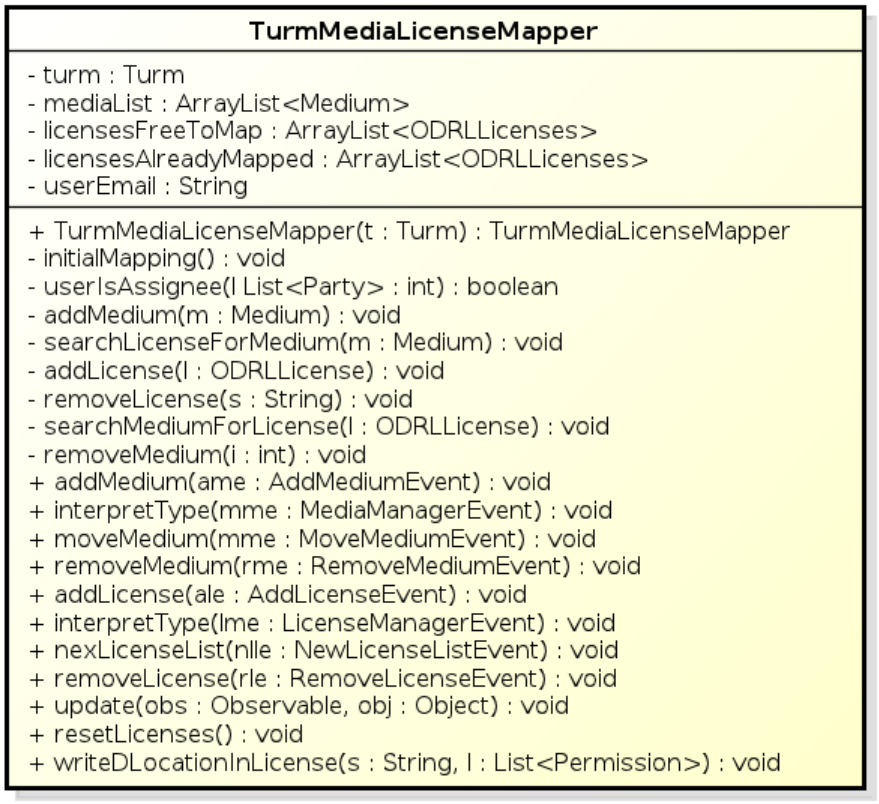


Figure 4.15: a part of TURM's class diagram related to the media manager: the media license mapper

The userIsAssignee operation is simply an operation to check if the user is assignee among a list of assignees. The addMedium operation works like the initialmapping. It creates a new media which it adds in the media list and then search a license for it. To look for a license, the user browse through the licenses free to map to check if there is one matching performing the same kind of tests than the initialMapping does. The addLicense operation works dually: when adding a license, the class possess operations to look for a possibly matching media. The operations to search a media or a license are searchMediumForLicense(ODRLLicense license) and searchLicenseForMedium(Medium m). There are also removing operations which work the same way (suppressing the media or the license, then notifying to the other part that the link doesn't exist anymore).

### Public operations

The operations of this sections are the one triggered by the GUI's events. They are the implementation of the supertype's operations. THe first one is simple and only adds a medium to the media list. It simply passes the medium linked to the medium event to the internal addMedium operation.

The interpretType operations is called whenever the MediaManagerEvent gets fired. It simply tries to guess of which type the manager event is (it can be addMedium moveMedium or removeMedium). If one of those 3 operations are recognized, the public related operation is used. It also work the same way with LicenseManagerEvent. The move medium uses a private operation wrote afterwards which simply find the license related to a certain media

and updates its DLocation. the removeMedium, addLicense and removelicense operations work like the addMedium operation.

The only operations left are newLicenseList and update. The update operation only works if the user's email address is updated. the newLicenseList is called to remove all the previous licenses and create a new license mapping. Instead of calling the constructor, it calls the resetLicenses operation and then calls the initalMapping operation. The resetLicenses operation simply clears the two lists and sets the licenses of the media in the media list to null. It then add all the license from the TurmLicenseManager to the licenses free to map attribute.

### 4.8.4 TurmMediaManager class

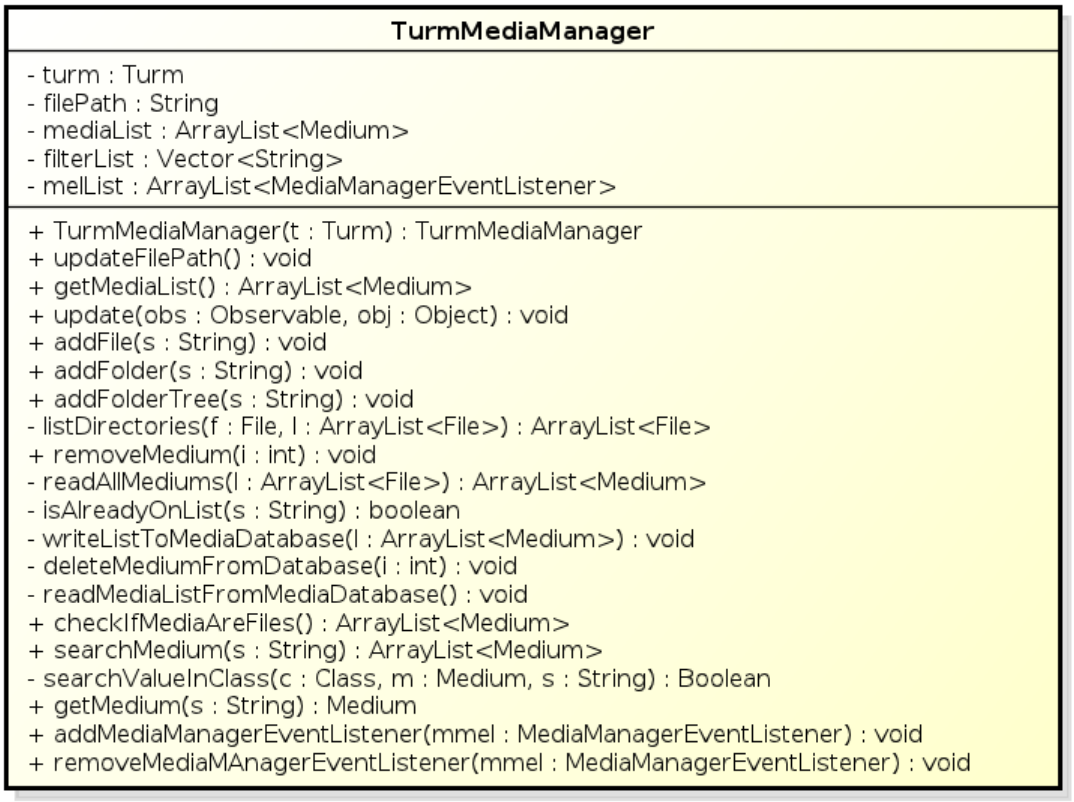


Figure 4.16: a part of TURM's class diagram related to the media manager: the media manager

This class is the main class of the media_manager package. As its name says, it manages the media. It also implements the Observer class. As in the TurmMediaLicenseMapper there are public and private operations. We'll go through them and we will also describe the attributes.

The attributes are quite easy to understand. The first is the instance of Turm (the core class). it is required by the media manager in order to observe it and update any changes related to the media. The second is a deprecated attribute : the filePath. The third attribute is the list of all medias currently in the TurmMediaManager. The fourth is a list for filtering data types. If a data type is in this list, it will not be added to the media list of the media manager. The last attribute is the list of MediaManagerEventListener's.

The constructor is quite simple, it just instantiates the Turm attribute with the parameter and then calls readMediaListFromMediaDatabase() which retrieve all the media in the

database and fetch them to the mediaList. While creating the mediaList, the fireMediaManagerEvent is called to notify the observer that there is a new media in the list and that it should be monitored.

The first operation is in a way deprecated. It is not used anymore in the current implementation of TURM, but as soon as the peer to peer plug-in will work, those operation will be called again. This operation is meant to update the file path. It calls for the incomingFilePath variable in the ConfigurationManager class. As the files do not have to be anymore in a specific directory, this operation is now deprecated from the current use of TURM.

The operations getMediaList(), addFile(String s), addFolder(String s) and addFolderTree(String s) are quite trivial and won't be explained because they are not subject to interesting lines of code. It must be said however that the three "add" function perform an operation with the database. The database contains all information about the files that were used at the last running of TURM. During the addFolderTree operation, the listDirectories(File f, ArrayList<File> l) is called. The listFiles(ArrayList<File> l ) is also called for the same purpose. It is just used to recursively browse a folder tree and return the list of all the folders in it. The removeMedium operation browse the database for a specific media (whose identifier is a parameter of the operation) and delete it.

For the database operations, there is a database manager who deals with those operations. However as it is located in the core class, the media manager can access it and make a call for media addition or removal. That is why the deleteMediumFromDatabase and the writeListToMediaDatabase are for.

The update operation works whenever the filter list is modified. The Media Manager has to update its filter in order to keep the current configuration of TURM.

The last operation is readAllMediums which is called each time a file is added. This operation is the converter from file to Media type. It simply takes a list of files and creates Media type out of them.

## 4.9 Media subpackage

This subpackage only contains 3 classes which depend on each others: Medium which is the super class of AudioMedium which is the super class of MP3Medium. Those classes won't deserve a complete diagram as well because they are only "data representation" classes. By data representation we mean that they were only created to contain data and their operations are only getters and setters.

### 4.9.1 Medium class

The Medium class is the most high level class of the package. It extends observable class because it is meant to be observed by the GUI. The goal is to be accurate in the GUI about the data regarding certain songs. As a reminder, TURM only manage MP3 files for the moment. We will however go through the three classes and see which additional information are in the subclasses.

At first there are two deprecated attributes : name and Filedir. Those attributes were initially used to identify the file but they were merged into one attribute : the absolutePath. We must however note that the two attributes are still used by the cup plug-in. The absolutePath attribute as it says, refers to the absolute path of the file.

There is in the database an unique identifier for each song in the database. the attribute uniqueIdentifier contains the value of the identifier of the database. It is not equal to the payload's hash-value.

The data type of the file is also integrated into an attribute (dataType) and the last attribute is a license (which is of ODRLLicense type).

Except from the constructor which is an empty operation, all the other operations are meant to set or to get an attribute.

### 4.9.2 AudioMedium class

The AudioMedium class is the direct subtype of the Medium class. Just like its super type, it doesn't involve many "advanced operations". Except from the readMetaDataFromFile() which is an abstract operation, the purpose of this class is the same as the one for the Medium class except that its attributes are refined to audio media.

The attributes refer to the meta data of the audio file, that's to say every information but the audio content. It contains the artist's name, the title of the song, the album the song was taken from, the track number of the album, the year the album was published, a comment about the audio file, the genre of the music and the duration of the song.

All the types are String which is an easy way to deal with information with the GUI. We don't know if this choice was made because of the GUI or for other reasons, but that's the only one we found for creating String attributes out of year and track number. All the operations are meant to return a value or set one in the attributes of the class.

### 4.9.3 MP3Medium class

The MP3Medium class is the most low level class of the package meaning that we don't intend to see unimplemented operations here and that we are supposed to see much more code. We can start by saying that like it's super types, this class also has a bunch of attributes with their getters and setters. We will though explain them with greater detail than the other attributes because they require a bit of MP3 structure knowledge.

The first is the MPEG version. As there are no specific format for MP1 and MP2 files, this part of the header give user information about the MPEG version. It can take the values from one to three. The layer attributes refers to the audio layer of the file. For the same reason it can also take values from one to three.

The isProtected, isCopyrighted and isOriginal booleans are meant to give the user information about the copyright the protection and the originality of the file.

The bitrate and sampleRate values are meant to give information about the quality of the file. To keep it simple, the higher they are, the better the sound is. There is a human fence as for the video quality. For the bitrate it is 192 Kbps and for the sample reate it is 20Khz. Note that human ears cannot get the difference from a bitrate of 120Kbps and a bitrate of 192kbps.

The isPadding attribute tells the GUI whether the MP3file has padding bits or not. Most audio files use 418bytes frames and some use 417byte frame. the goal here is to tell if the frame is padded or not.

The isPrivate bit is purely informative. The channel bits are 2 bits to tell what channel mode the song was designed for.

- 00 : stereo
- 01 : joint stereo (stereo)
- 10 : dual channel (2 mono channels)
- 11 : single channel (mono)

The emphasis attribute is sedlom used in MP3 files, it indicated the decoder if it has to re equalize the sound after a noise supression. This is not the full information about the emphasis attribute but it is all we need to understand it.

The getters and setters regroup the getters and setters from the class and from the super class, that's to say from the AudioMedium class.

## 4.10 Package license_manager

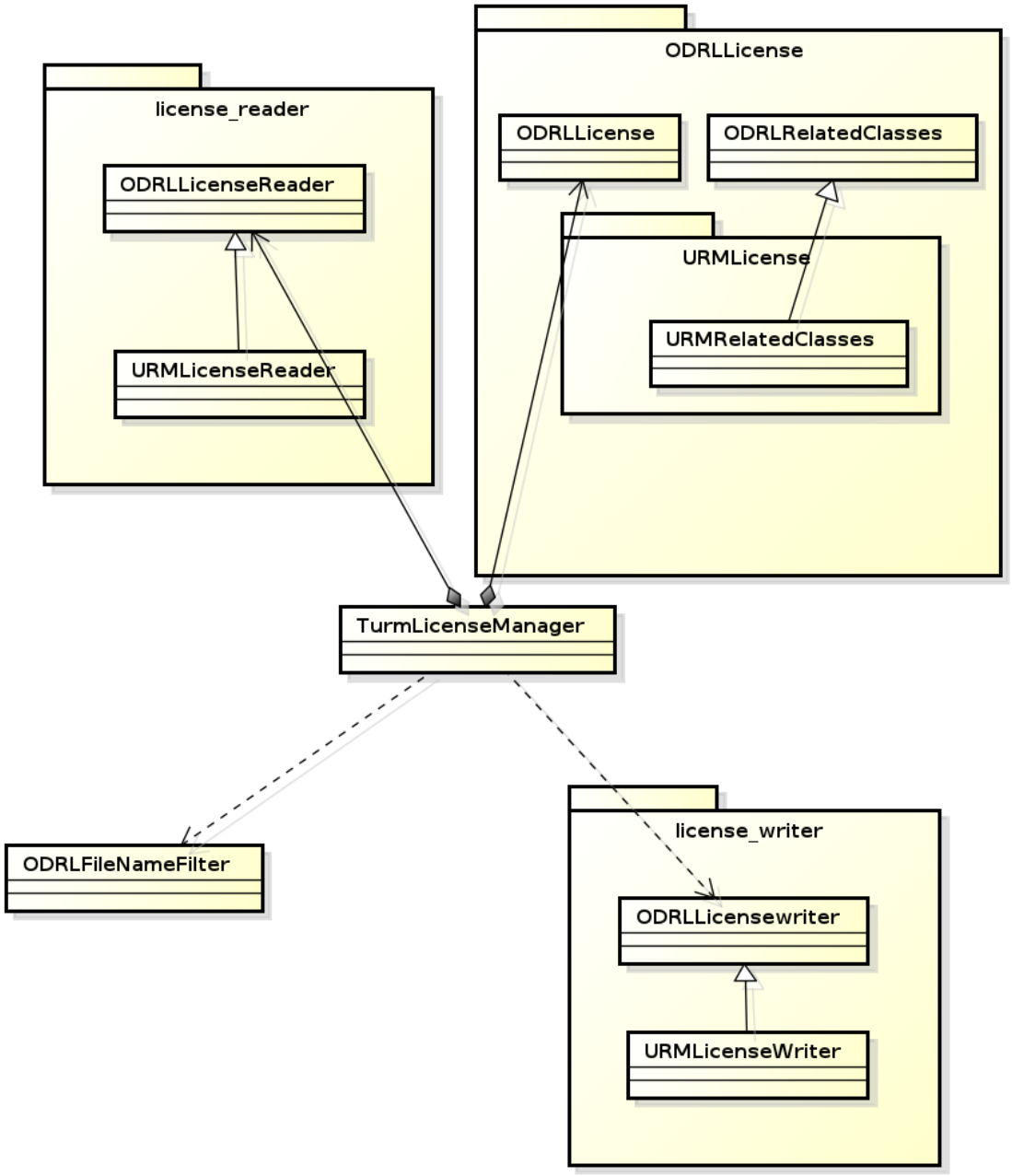


Figure 4.17: the generic class diagram of the license_manager package

The license_manager package contains as it names says the operation related to license management. We must say that the class diagram quite differs from what really stands in the package.

First, the ODRLLience package and the URMLicense package implements the ODRL and the URM license respectively. It was thus not useful to remind the reader the previous chapter. The only useful thing to know is that the URM licenses are subtypes from the ODRL licenses (at least in the program). It is not clearly specified if URM are a direct

subtype of ODRL but TURM considers that to be true.

Second the DBManager classes. There are two DBManager classes in the license_manager package. Those are both deprecated but it is not mentioned in the documentation. As the class diagram was complicated enough we didn't mention that they were there by opposition to deprecated operation and attributes which are mentioned every time. The OdrlFileNameFilter is also a deprecated class. It is not mentioned in the documentation but the call hierarchy of its only operation is empty.

Third, as for the media_manager package, the amount of operations and attributes was so high that we erased them for this diagram to show a general structure. Also the classes related to the license themselves (ODRLLicense package) were also removed in order to keep the diagram on one page. Moreover the dependencies between the license classes were explained in details in the previous chapter.

The TURMLicenseManager class is composed of 2 arraylists of ODRLLicense classes and is also composed of one ODRLLicenseReader. It does not have an ODRLLicenseWriter but depends on it as some of its operations call for one. It also has dependencies over ODRLFileNameFilter class. Of course it has many other dependencies but our goal is to show the reader that every class of the package belong somewhere and is linked somehow with the "main class" of the package.

### 4.10.1 TurmLicenseManager class

TurmLicenseManager is the main class of the license manager package. We'll first go through its attributes and then try to browse its method logically.

The first attribute is the instance of the main class of the application : Turm. It is also used for the same purpose that's to say configuration manager access (to get the default license file path, to get the database manager and so on). Then there is a getsubfoldersFlag which is a boolean. It is used to tell the application to browse for more license files in the sub-folder of the license file path if it is true. The folderPath string is the default location of the license files.

The two next attributes are there to make a distinction between the owned licenses and the export licenses. It is possible with Turm to set an export license directory where all the export licenses would be stored. That is the function of myLicensedirectoryPath and exportLicensefirectoryPath. Following the same logic, there are two lists of license files, the owned licenses (myLicenseFileList) and the export licenses (exportLicensefileList). Still following the same logic, there are two lists of licenses : one for the owned ones (myODRLLicenseList) and one for the export ones (exportODRLLicenseList).

The ODRLLicenseReader is the tool used to visualize licenses. As we said earlier, the ODRLLicenses are XML files and the ODRLLicenseReader provide operations to extract the content of these XML file in order to build an ODRLLicense (the class). the urmMode boolean as its name tells is set to true if the license to manage are URM licenses. There is also a database manager and a boolean which is set to true if the licenses are loaded from the database at first. At last, there is a list of licenseManagerEventListener.

The two first operations are constructors. The first is only used if there are no specific directory path for the license (the default one must be used). It simply calls the second constructor with null as the path value. The second constructor instantiates the database manager, the folder path, calls the initalize() operation and then adds an observer to this. The initialize operation sets the directory paths correctly and then, if the loadFromDatabase boolean is true, loads from the database the licenses. Once loaded it calls the updateLicense() operation. This operation works with the updateLists operation in order to synchronize the database and the license folder: it adds to the database the licenses created, it removes from database the files that are not anymore in the folder and so on. If the loadFromDatabase boolean is false then the initialize() operation calls the


Figure 4.18: A part of the license_manager package class diagram : TurmLicenseManager

importFolders() operation. This operation is meant to get all the licenses from a specific directory. It calls the getLicenseFileList to get all the owned licenses and all the update licenses. Once the files are gathered, it calls either for an ODRL constructor or for an URM constructor according to the urmMode boolean value. When the licenses are loaded in the application as license objects, the initialize() operation calls deleteLicensesFromDatabase() and storeLicensesToDatabase() in order to refresh the database with the loaded licenses.

The createExportLicense takes three parameters : the hashvalue of the file, the assigner's e-mail and the assignee's e-mail. It then creates a license with those parameters as we saw in the previous chapter.

The operations fireLicenseManagerEvent and addLicenseManagerEventListener are GUI operations : the first to inform the eventListeners about a new event and the second to add a licenseManagerEventListener to the class. the getMyODRLLicenseList() operation returns the licenses owned by the user and the usUrmMode() returns true of the licenses are URM licenses. The printToConsole operation seems to be a debug operation used to send a console message with "TURM License Manager" prefix. Is is the same for the runTest-Code() operation which is a debugging class who should be deleted (there is an explicit note in the comments). The last operation of the manager is the saveLicenseToFile operation which as its name says is an operation to create an ODRL or URM license file under an

XML format.

### 4.10.2 ODRL and URM license reader and writers

In order to not confuse the reader, we decided to not describe these sections in details. those classes are in deed very important but there is not much to say except that they provide the implementation operations to read and create ODRL and URM licenses.

As those two concepts have been explained in details during the previous chapter it wouldn't do anything but confuse the reader to explain what is the detail of those classes.

## 4.11 Db_manager package

This section won't be started by a class diagram because there are only three classes and that they are not strongly related to each other: There are no compositions or hierarchy between the classes. Something else to say about the database is that TURM uses hibernate as a framework to manage the database. As the database has not been created to express relations but more to store files, a schema is not needed. We will just browse through the data we can store.

### 4.11.1 Storable data

There are two kinds of data who can be stored actually : files and licenses. By "files" we mean information about files such as their location their hash value, their type and an identifier. For MP3 files there are also all the meta information we discussed above.

For the licenses, again there are all the information about licenses in the database. That's to say : the rights available, the assets on which the rights are defined, the Duties the user has to perform, the constraints on the licenses, the Parties which are the assigner or assignee for the license and the license right expression. The only shortcoming of this database is that it could have been designed with foreign keys to have a more secure database.

### 4.11.2 DBManager class

The database manager is a class with two kind of operations : read and delete. This is possible to read Media and ORDLLicenses from the database. It is also possible to store objects or list of objects and finally delete a line or a whole table in the database.

We won't go in further details because all the operation involve the use of the hibernate driver which hides its internal operations from the user.

### 4.11.3 HibernateUtil class

This class is just an initializer for the database, it creates and grant access to the session used for the database operations. The only thing of interest to say about this class is that it uses the singleton pattern in order to be sure that only one instance of the session is kept during the execution of the application.

### 4.11.4 SQLiteDialect class

This class is uncommented and there are no clue about it's use. However as it wasn't made by one of koblenz's developers, we believe it is a class which is used by hibernate to communicate more easily with the database.

## 4.12 Code cleaning

During our internship, we first had to make sure Turm could work before doing any plug-in operation. There were several bugs we found who can be mentioned.

### 4.12.1 path issues

There have been a lot of path issues because everyone wanted to use its own and personal path, when we got the program from SVN, we realized all the configuration files were in the project, but the configuration folder was set to a Windows like path. As we were working on Linux this seemed quite strange and when we found the variable responsible for this, we simply put the relative path to the application which will no longer lead to path problems as everything is stored in the application's folder.

### 4.12.2 Hibernate issues

There have been some path issues due to hibernate too. When included to an application, Hibernate simply use the root repository of the application to get its configuration file. We tried to find some explanation in the Internet but as Hibernate is poorly documented, we could not find a way to change the configuration folder.

The problem is that when trying to export all libraries into a plug-in to be a bit more clean, hibernate takes a whole new folder (The user's home repository) and it is then not very effective to put configuration files for an application into the user's home folder.

### 4.12.3 Comments

The code was generally well documented except from certain parts which were a bit improved. The database package wasn't improved because as we said in the previous subsection, Hibernate even if useful is very poorly documented.

### 4.12.4 Deprecated classes

There were deprecated classes and even deprecated packages which we had to clean. This was not a hard work though because the majority of them were pointed as deprecated. The only difficulty remained when we realized that some operations were deprecated but used by the non implemented CUP package. This could lead to some issues when CUP will be fully integrated into TURM.

### 4.12.5 ConfigurationManager

A huge shortcoming of Turm is that the configuration manager is an external library and not a class of the core. As there are hard coded values in there, it would have been useful to be able to access them.

## 4.13 Conclusion

This chapter allowed us to get a more concrete view of TURM. We started with a vendor description to progressively go to lower layers of the application in order to get a deep understanding of all the classes and why they were here for. As our main duty was to study the plug-in conversion for this application it sounded useful to describe it in details to get the reader to know what were the dependencies between the classes and what would be a good thing to export as a plug-in or not.

We also got an overview of the pre-work to do with Turm in order to make it runnable and we saw some shortcomings of Turm we had to fix. In deed as the program had been updated by many people who didn't focus on the execution of Turm in general but more on the implementation of a feature, Turm had not been run for over 3 months before we opened it. It was not trivial to get everything working even if there were no huge flaws. The problem mainly came from the fact that the paths to the configuration files were set only for one user leading to a problem for other users to run TURM without editing these paths.

In the next chapter we will get a focus about how concretely implement a plug-in infrastructure for TURM and what are the parts which are more suitable to be a plug-in and which are the parts we decided not to export as a plug-in.

# Chapter 5

# Plug-in implementation solutions

## 5.1 Introduction

So far we have presented all the tools we needed. We explained what were the different solutions to implement a plug-in framework and we chose the one that seemed the best. As a reminder, we chose OSGi because it seemed more documented so the future developers will be able to get familiar with OSGi quickly. We also chose OSGi because it was a framework by opposition to JSPF which was only a library and because it could handle plug-in management more easily than the other frameworks we had.

In order to describe TURM, there was also the need of a chapter explaining its purpose. We thus wrote a description of URM following its creator's documentation and conferences and also ODRL according to the community responsible to make it a W3C norm.

We also discussed in greater details the functionalities of TURM and browsed the class diagram. We saw TURM as if it was sold by a commercial and then we saw how each functionality was implemented and what was the class architecture. We also saw some difficulties encountered during the cleaning of the code when we started working in Koblenz. Now we will discuss the plug-in implementation.

First we will describe the specification the university of Koblenz gave us about the application. Starting from there we will try to figure out why a plug-in solution might be better than a straightforward implementation. This part will be more intuitive than scientific in the sense that we will less focus on the program's capabilities and structure, but more about what our client wanted from us.

Then we will describe how a plug-in implementation is possible. In order to not only pick a solution, we will describe several possible solutions and discuss them in order to figure out which of them is better in the current case.

Then we will describe the changes that had to be made for TURM, we will explain what has changed since the straightforward implementation and describe in an abstract way the relations TURM has with OSGi now.

## 5.2 Relevance of a plug-in implementation

Before discussing how to implement TURM in a plug-in way, we must first discuss the need of such implementation. There were two very positive points that encouraged us to develop the plug-in implementation. We will discuss them and explain some shortcomings of a plug-in implementation in that case.

### 5.2.1 First reason : fast and easy evolution and customization

As our client in the university of Koblenz told us, TURM is really a toolkit. It thus must provide the user with a lot of tools which could be enabled or not regarding the user's preferences. We will take the example of Payload Extractor.

As there are many extensions and that new extensions are still written these days, if TURM becomes a reference implementation and is actually used by a lot of users, it will then have to be able to read the most extension possible if not all of them for the media. This means that the developer team will have to change TURM every time there is a new media extension. This means browsing TURM's code and implement the solution we want. Also, this means that if we want to avoid a complicated implementation we must only have one payload extractor for each extension. But it might be hard to get a payload extractor that would be suited for all the need of TURM's users.

With a plug-in implementation, There is no need to mind the whole application but only the part that is needed for the payload extractor extension. That's to say, the only thing needed is the interface TURMFeature in order to see which operations have to be inherited. Also, there can be more than one payload extractor implementation for each extensions. The plug-in will simply have to be stopped. In that way, without gaining in complexity, TURM would be more suited for the need of its users.

Also, as the knowledge needed to create a plug-in is less important than with the straightforward version, it might be possible for every computer scientist with a Java base to write their own plug-ins.

In a pluggable application, the evolution and the customization are far more easy to do than with a monolithic version. This is one of the motivations to make this plug-in development.

### 5.2.2 Second reason : development of new features

With a toolkit, sometimes it might be useful to add a tool that wasn't meant to be implemented when you first created the toolkit. This was another concern of our client which was not sure whether or not to develop new features. Being wise, he decided to plan the eventuality of new features even though he did not have clear plans for them. His plan was to use TURM for example to be a base for student bachelor and master thesis who would have to implement new kind of features in practical thesis.

The problem is that if we keep TURM with its straightforward implementation, as the features are added, the code becomes more and more complex. We will not discuss in this thesis the number of implementations that clearly respect programming guidelines but we will assume there are a lot of large application which are not designed according to those guidelines resulting in an endless list of patches to correct bugs. Knowing that bachelor student might not be experts in Java programming, it might be unsafe to leave them the whole code. Also, it might be hard for them to understand the entire TURM application in two or three years.

With a plug-in development, we resolve those two problems. When the student will implement his solution, he won't be able to write into TURM's core and so even if he fails to implement his solution, the integrity of the code constraint will not be violated. Also, if TURM gets complex, all the complexity will be relocated in plug-ins and not in the core itself.

We must though underline the fact that if a whole new tool is written, the developer will have to create the interface in the main plug-in otherwise it will not be possible to create the OSGi service.

The general idea here is to get the student to be less eager to write everything in a single package and to force him to think in a clean and structured way.

Again, like the first reason we invoked, if a student have to create a new feature of an existing kind (for example a payload extractor for WAV extension) he will not be given the core and thus will not be capable of introducing bug in it.

### 5.2.3 Shortcoming : management of plug-ins

The main shortcoming with a plug-in implementation is that we introduce the need to manage plug-ins which is not what was in the original TURM's code. This is a complexity increasing and every developer who will write a plug-in will also have to take into account its management. nevertheless, one of our clients requirement was the easy customization forcing thus the future developer to write a "feature manager" even if it is not in a plug-in form.

If a user doesn't need payload extractors for videos, then he should be able to deactivate the option "payload extractor for video extensions management". And this will be implemented more easily with a plug-in structure than with a straightforward structure. This because all the plug-in management can be located in one place.

It has the advantage of managing very easily the different plug-ins even of different kinds but this also implies a single point of failure which is not safe. However as the management of plug-ins is ruled by this single point of failure, the bugs regarding plug-in management will be easily found and as we saw earlier, OSGi's service tracker is not very complicated to understand and use.

### 5.2.4 Choice between straightforward and plug-in program

It is obvious regarding our objective that a plug-in implementation is the best choice. We must underline now that even if some plug-ins seemed a good idea, sometimes it was irrelevant to make a plug-in out of them that being due to the lack of evolution in the sector.

For example, the hash calculators looked like good plug-ins. But the fact is that there are no really need for a plug-in development because there are very few hash calculators and they can fit for everything.

One hash calculator can be enough for the whole application. By opposition, a payload extractor for the MP3 extension will only fit for the MP3 extension and nothing else.

By the time we were specified that there would only be one hash calculator we already had made a plug-in out of it. We kept it that way with the same thought for prevention that our client had: we might in a far future implement a new hash calculator but for the moment this one is enough. That means that if one day a developer wants to develop another hash calculator, the service handler would already be ready for him.

## 5.3 Discussion over the possible implementations

As we said in the introduction, we explored three ways of implementing an OSGi service. This is separated into two point of view : the location of the service and the classes shared with the plug-in.

### 5.3.1 Service location

The first way to implement a service is on the external plug-in itself. That's to say, the service has to be available when you start the plug-in. The core can then register itself as a consumer of the interface. This is intuitive and the most obvious solution. There will be as much services as you have plug-ins. The shortcoming of this solution is that the list

of available and registered plug-ins will have to be refreshed constantly. We will go into greater details in the conclusion of this section.

The second way is to make a service from the core on which the plug-ins can register when it is started. With that approach it would be more easy to manage the services as you do not have to browse for new services, they will join and subscribe themselves at start. The negative points with this approach is that it becomes difficult to implement. As it is said, a service can be seen like an implementation of an interface. Doing the service the opposite way would mean that the plug-in would have to possess the interface which the core plug-in would have to implement. This is possible, but it would not be clean, and it would be more complicated than the first solution.

We will thus prefer the first solution which is more clean because all the services are autonomous. Besides, it might seem that writing a service contains all the "OSGi" code, but when you register a service, both parts contain OSGi code and it is clearly easier to create a PayloadExtractor interface (which already exists) and to create an implementation as a service which would be "PayloadExtractorMP3" as it is the case.

Also at first we did not know how the service tracker worked so without it, we would have had to create some sort of manager which would have constantly browsed the framework for new services. This is not the case here as the service tracker handle all the services management. The implementation will be presented later.

### 5.3.2 Classes shared

The second point of view is the amount of classes shared. We have numerous solutions but will take the two extremes: sharing everything and sharing nearly nothing.

When everything is shared, we mean that you make each service able to access the core classes and manage their business code knowing that they have full access. The problem of this solution is that the plug-in is given too much control which should not access the whole application. This might result in inconsistency and piracy. As a plug-in is given access to the main classes, it can do whatever it wants. The happy scenario in that case would be an unexperienced developer who introduces a bug in the main application. The worse scenario in that case would be an experienced developer conscientiously introducing an exploit into the application. The second shortcoming of this solution is the refinement of the design. By that we mean that giving full access, is like having global variables in an application: it might be useful in certain case, but writing an application with only globals wouldn't be optimal.

In the second solution, we would only share the strict minimum, often nothing. The service is an implementation for a specified interface of the main plug-in and therefore should not access anything but be accessed by the core. This solution was chosen because the first developer team had already designed the application that way. We just used their work in order to extend it to a real plug-in application.

### 5.3.3 Best choice for a plug-in implementation

We identified three possibilities to implement our solution: the first would be to make the service in the external plug-in and we would not share any classes, the second would be to make the service at the edge of the core plug-in and not share any classes with the plug-in and the third to make the service in the core package of the core plug-in sharing TURM class which is the main class.

The third solution was directly rejected because it seemed badly structured. However, after a discussion with our client, we realized that the core was implemented with one class responsible for all the other with the thought that if one they there were plug-ins, they would be given the right to use that general class. This is a structure we decided to

let that way but that we didn't use. It seemed more secure to give the minimum. Also, knowing that developers would have to implement a plug-in, it was best to give them the least information possible in order to not complicate their task.

The second solution was rejected as well because it wasn't optimal. We already had a payload extractor interface and an implementation. The easiest way was to take the implementation out of the main plug-in and to make a plug-in out of it. Furthermore it was as clean as it was easy because the plug-in is only composed of two classes : the implementation of the payload extractor and the activator.

The first solution seemed and still seems the best. Thanks to the service tracker, we do not need to manage all the services. Also, the service tracker possesses operations to shutdown and restart services.Thus, making a small interface who would allow the user to check or uncheck the features he does not want would be very easy. And finally, it respects the OSGi description of a service, it is the most intuitive implementation and it also has the advantage of being clean and relatively secure. We will discuss about the security in the next chapter. Now we will present our solution explaining what has changed in the application of the last chapter.

## 5.4 TURM with plug-in implementation

To begin with, we will describe the two plug-in we made. They will be described in details like we described TURM's classes. Afterwards, we will present of course the main package, more precisely what changed.

Then when all the new classes will be presented, we will explain the relation TURM now has with OSGi. This will familiarize an unexperienced reader to the links the classes now have with the framework or more precisely, which classes have communication with the framework. This will be presented as an UML class diagram but will not respect UML's semantic. It will not be a problem though because this diagram will be fully commented.

Finally, we will discuss about some features which looked good as plug-ins but were not implemented that way because there were problems. We will detail these problems and explain in what way they prevented us from developing plug-ins.

### 5.4.1 Service PayloadExtractorMP3

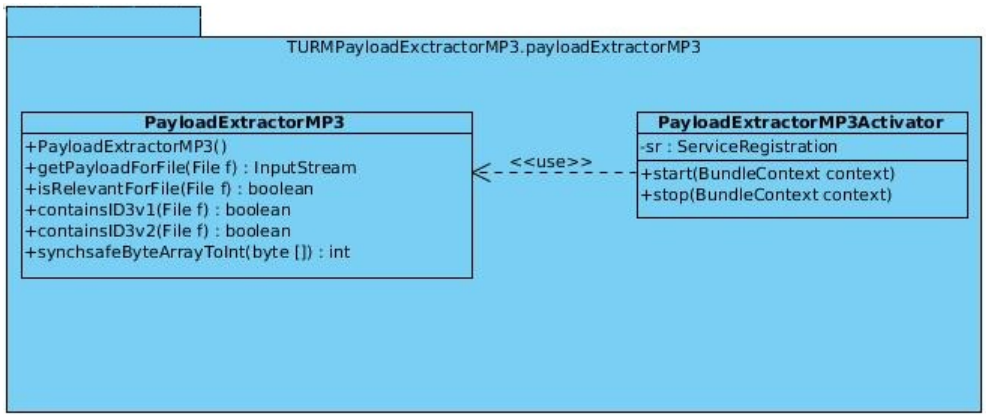


Figure 5.1: The PayloadExtractor plug-in implementation

In the main application, nothing has changed except that the PayloadExtractorMP3 class was removed. The reason we did not have to modify the code is that there already was a plug-in issue handler which could manage the display of extensions which are not managed by the main application.

The class PayloadExtractorMP3 remains the same.

We created an Activator. This activator is composed of one private attribute which is a ServiceRegistration. This variable is responsible for registering services in the framework. In the start operation, we first instantiate the PayloadExtractorMP3 class, then create a dictionary in which we put the key/value couple which is supposed to be handled by the service tracker. Then we register the service and as a feedback print that the service has been registered.

The stop operation is much more easier as we only use the unregister() operation fro the ServiceRegistration class. Then again as a feedback we print that the service has been unregistered.

### 5.4.2 Service HashCalculatorSHA-1

This is exactly the same reasoning than for the payload extractor feature. The only problem is that as TURM stand for "Tools for Usage Rights Management",without a hash calculator, it is impossible to write licenses. That being one of the main TURM features, the hash calculator registration was hard-coded in the core class. We had to find a way to implement it more properly in the service tracker. That will be explained in details later.

As the functions are the same, we will not discuss the HashCalculatorSHA1 activator. This diagram was made only for documentation purposes which is what the client also wanted us to do.

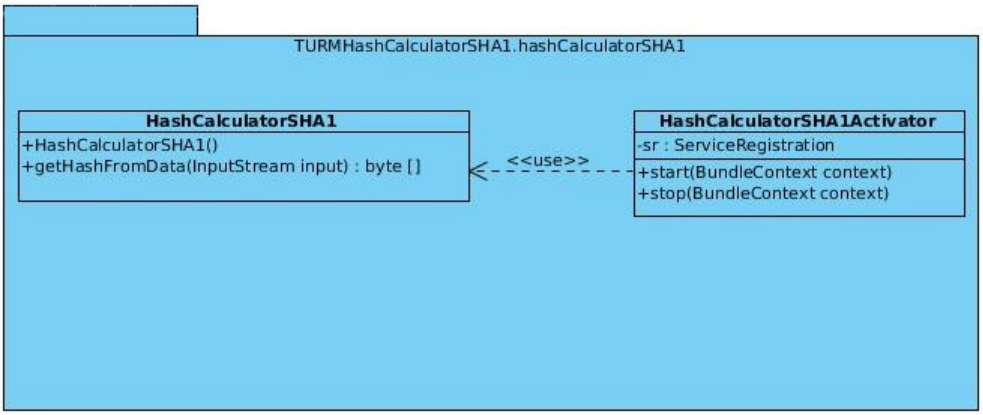


Figure 5.2: The HashCalculator plug-in implementation

### 5.4.3 Core implementation

These are only the two plug-ins we made. But further than that, we created TURM's service tracker which was thought to be able to support more than these plug-ins. At the moment, developers are working on CUP interfaces. As a reminder, CUP is a peer to peer implementation for TURM and as they wish to make some different implementations or peer to peer interfaces, they will be using OSGi and our service tracker.

The TURM activator is simply the main class of TURM which has been restructured in order to be started as a plug-in. The only addition from the main class is an instantiation of the service tracker.

The service tracker it is a bit more interesting. The constructor remains the same as the super so we just make a call with a filter we instantiate ourselves. The getTrackerFilter(BundleContext context) operation allows us to set a filter ourselves. Technically this filter is simply a string, but the way we create it is a bit tricky. We explained earlier that

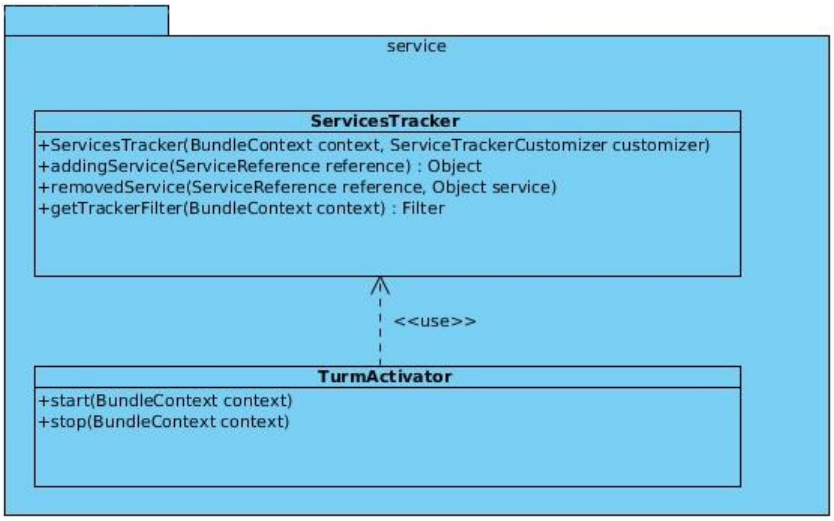


Figure 5.3: The core plug-in implementatoin

it used the reverse polish notation and that we already have added a dictionary with each of our plug-in.

At the moment these dictionaries only contain a variable with "iface" as a key and the kind of interface the service implements as a value. We chose to implement all the service in a single service tracker, but nothing forbid further developers to split this service tracker in order to make service trackers for each features.

We did not at the moment see the point of implementing this solution because there are only two kind of plug-ins available (payload extractor and hash calculator).

Then as the main class directly instantiated the payload extractor for MP3 and the SHA-1 hash calculator, we had to find these instantiation lines, and remove them.

They were added in the service tracker into the addService operation.

The addService operation is simply a big switch. It browse if the the service that was just registered was an instance of a class that is managed by the service tracker. So if this service tracker notices that a new service is logged,it will first search for an iface value in the dictionary. If it matches the filter, it will try to add the service. In order to do so, it will compare the class with the HashCalculator class and the PayloadExtractor class. If the service matches one of the two, it will be instantiated as such in the main application. By instantiated, we mean that it will be added to the list of payload extractors or hash calculators. As a reminder, the services possesses already instantiated classes.

### 5.4.4 Relations between TURM and OSGi

Now that we described the new implementation, we will show the relation the program has with OSGi. To begin with, the OSGi class that is in the upper left corner does not exist. It can be assimilated to the whole OSGi framework.

This framework can have relation with the three activators mentioned above. We say can because of course, you can start the framework without any plug-ins. If you start one of the two feature services, they will just log in and wait to be used. To do so, they will instantiate the payloadExtractorMP3 class or the HashCalculatorSHA1 class. There can only be one instance of each plug-ins at once. This is a choice we made because we did not see the point of having two payload extractors or hash calculators that would be responsible of the same thing.

As TURMActivator is instanciated, it is still composed of an instance of TURM class which is composed of two managers each responsible for payload extractors or hash cal-

culators. But the activator is composed of one service tracker which is used by the OSGi framework to log the services. By "use" it would maybe be more appropriate to say that the framework will notify the service tracker whenever a service logs in AND match its filter. But in order to know which filter is set for the service tracker, the framework must be able to manipulate it. The service tracker of course uses the two TURMFeatureManager classes.

This was a short but easy to understand diagram in order to understand how everything works right now in TURM.

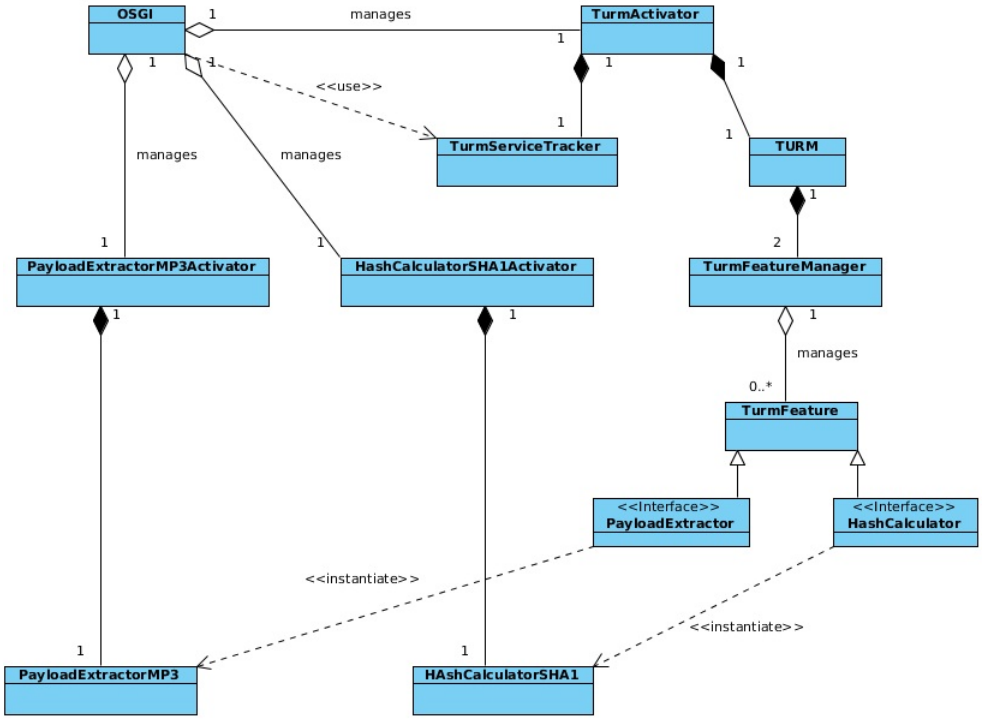


Figure 5.4: The relations between TURM and OSGi with the plug-in implementation

### 5.4.5 Not implemented plug-ins

As we described the hash calculator, we figured out that it might have been useful as well to make out of the license readers and writers a plug-in. That would render ODRL much more flexible if it could be represented of several ways and if it could handle these. As working on ODRL was not our goal in this thesis, we more focused ourselves on providing a plug-in support for this idea.

The problem is that, as we said, much developers have modified TURM since its first developer. And all of them did not have the plug-in focus we had, including the developer of the license_manager package. This resulted in cyclic dependencies.

What we mean by cyclic is that with OSGi, A plug-in whose requirements have not been resolved cannot be resolved itself. So if plug-in A needs plug-in B and plug-in B needs plug-in A, it is not possible to implement any of them as plug-ins and they have a so strong dependency that they should remain in one plug-in only.

In our case, the license manager needed the core to work and the core needed the license manager to be instantiated. Those needs made it impossible for us to create a plug-in. Again, we could have modified the whole license manager in order to fulfill our needs, but when discussing with our client, we realized that the license reader and writer were

good that way and that they were far from thinking about creating new ones.

We also wanted to implement a plug-in which would have contained all the libraries. This seemed like a clean way to work instead of putting in every plug-ins the libraries needed, we could have had a library plug-in. It would not have posed cycles problems because this plug-in would have been needed by many plug-ins but it would not have needed another plug-in to be resolved.

This was however given up because some libraries need to be in the plug-in they are used for otherwise their configuration path is set to the user's home folder.

As we did not want to pollute the user main repository for this application and as we wanted the application to remain standalone, we decided to leave the libraries in the main plug-in as it was the only one needing libraries.

## 5.5 Conclusion

In this short chapter, we first justified the need of a plug-in development for a medium sized application. As we justified the need of this development, we also explained why a plug-in infrastructure is better in specific kind of programs with specific user need.

We described our point of view and what were our angles to bring a solution. We did not prove that plug-in implementation is always better than monolithic implementation but we justified its existence in an application which has highly modular needs.

We then presented the three possible ways to implement a solution and explained why two of them were no fit for this solution.

Once we justified ourselves, we explained how we modified the application in order to make it more modular. Our great asset was that the developers had already a plug-in development in mind and so had already made an easy structure for plug-ins. We explained that nothing had really changed in the code except for the plug-in code which had to be added to the application. The shortcoming to that is that TURM used to have many developers which did not have the plug-in focus we had. This resulting on ideas of plug-ins given up because the code was too dependent from other parts of the core plug-in.

As a conclusion, we would say that our implementation far from being complicated and full of OSGi knowledge, has the advantage of being easily understood for a developer which is a great deal knowing that other developers are currently working on TURM and relying on our plug-in implementation.

What we did was more reorganization than pure creation, but we showed that we did not randomly move classes from one point to another. We showed that our structure was well thought, and that we did not focus on the first solution we though about but rather try to find several and present them to our client.

This was our implementation choice, but it is so because we convinced a client which was working in the computer science field that our solution was viable and optimal for his needs.

# Chapter 6

# Security issues for a plug-in infrastructure

This chapter will be a bit off-topic. The goal of this thesis was to discuss the advantages of a plug-in infrastructure in a large software package taking as an example TURM. However, even if the plug-in infrastructure provided by OSGi seems to be something that should be used for every large software package, we must identify some security issues.

These issues will not be developed exhaustively, because this thesis is not about the security of OSGi platform. Nevertheless, we would like to add to this thesis a security focus, developing a big issue and giving the basics about how to solve them.

We gathered together some articles which put in evidence some shortcoming of plug-in programming [ea05] [PHP08] [GJA08] [CCH07].

These shortcomings have not been sorted in the articles. That's to say they do not specify whether their solution is to prevent authentication, authorization or integrity issues. However we did sort the solutions proposed by their complexity and the level of security they offer.

Trivially, the more complex the solution is, the better level it offers (if implemented correctly). Unfortunately, many of the complex solutions are in fact just abstract designs who give you the intuition of the solution. There are no fully implemented solutions and this is why this chapter will not be detailed too much.

The purpose of this chapter will be to open the reader's mind to OSGi issues and give him a package of solutions.

We must underline that TURM does not need such specification features because it is not (yet) a sensitive application. But this might come in handy for readers who might be developing an highly sensitive plug-in application and who would be looking for a solution who would be easy to implement.

## 6.1 Problems identified

When we browsed the articles, we realized that they all tried to get a solution to the same problems, identification and authorization.

### 6.1.1 The identification problem

The identification problem is a problem that occurs when you have a need to restrict access to public to some sensitive data. For instance, let's say you want to reorganize your small company into departments. In order to know who is in which department, you have to identify everyone and to classify them. Then if you have to restrict access to a department, you will have to be sure the employee who wants to walk in is who he pretends

to be. Therefore you need an unique identifier per employee which can be done with several solutions.

This problem is handled by OSGi under the principle of PKI infrastructure. The main problem is that the PKI infrastructure is quite slow and so it is problematic because the user has to choose between speed and security. In the stated solutions, the researchers focus on a security solution which is not slowing the application too much. That's to say humans won't notice the difference between the secured application and the unsecure application.

#### The PKI infrastructure

The public key infrastructure is an infrastructure in which evey user can use keys to decrypt or sign data. Each user has two keys, a public key and a private key. The private key remains secret to the user and he uses this key to crypt or sign data. The public key is available to anyone.

This infrastructure works that way : user A has to sign an Internet contract. To be sure it is him, he uses his private key to sign the document. When user B receives the signed contract, he takes the unreadable signature (because it is encrypted) and uses user A's public key to decrypt it. If that works, it means the document was really signed with user A's private key which is only known by himself.

The advantage of this solution is that it is quite easy to sign a bundle with your private key and then let the receiver identify you. The problem is that it takes a lot of calculation time to perform these operations.

### 6.1.2 The authorization problem

One problem that is not handled by OSGi is the authorization problem. As you identified your user, the problem of the actions available for each user still remains. OSGi does not provide solution for that and so the designers have to couple OSGi with some sort of authorization component which will manage policies over users. A policy is a set of rules that say what which user can do. As an example of policy we could set "Employees who are not referenced as human resources cannot access the employees personal data".

## 6.2 First solution: using XML signatures for identification and XACML for authorisation

This section will explain in greater details the article of Hee-Young Lim et al.[ea05]: the XML signature. As we explained in Chapter 2, OSGi was chosen because it was the best for our needs, but we were also aware that we would not use it at its full power. OSGi can manage system resources for example network resource, printers, and so on. This is not yet needed for TURM but as a peer to peer client is about to be implemented for it, it might be useful to consider networking problems. To clarify things, we can simply imagine that OSGi abstracts the network in order to only consider that it has an extended framework. You can register your home network to a remote OSGi framework and then it will work just as if you were simply working in a local network space.

But as it is easy to develop applications that works through the Internet, it is also easy for malicious users to access the same program at the same network. This malicious user can then code a fake bundle and identify it as your own possibly granting him an access as if you were logged in.

As we said earlier the identification problem is handled by OSGi but the PKI is a bit too slow and this is why Hee-Young Lim et Al.[ea05] developed a solution which is as easy as the previous one to install but which is faster than it.

### 6.2.1 Motivation for the XML signature solution

To develop our solution, we first need to go further in the bundle authentication of OSGi. As a bundle registers, we clearly saw that you could put a lot of information in the dictionary in order to manipulate it with the service tracker. This is made because OSGi was meant to be as flexible as possible, but with this flexibility, it is hard to write a clear document specifying how clearly a bundle identifies itself.

First we must remind that bundles come in form of JAR files. Generally, to authenticate a JAR file, we use a mandatory access control based on the keys of the developer. The MAC (message authentication control) is created by using the private key of the developer and then to validate the access you must bring the public key of the developer which is how the PKI works as we explained above.

The problem is that to fulfill this identification, the platform does not possess enough memory and this is the motivation of the XML signatures solution. The scenario at figure 6.1 represents the identification process with the XML signatures solution.

In short, the intelligent pieces here, are to use an XML signature only to sign the manifest file which contains all the information about the bundle. Therefore it is the only thing that needs to be verified.

#### XML signature

An XML signature is a signature used to ensure the integrity of its target. This XML signature can be used either to sign entire documents or parts of them.

In our XML signature contest, this means you can sign either a complete jar file or only parts of it (like documents).

An XML signature has this basic structure according to W3C[com08]

```
<Signature ID?>
   <SignedInfo>
     <CanonicalizationMethod/>
     <SignatureMethod/>
     (<Reference URI? >
       (<Transforms>)?
       <DigestMethod>
       <DigestValue>
     </Reference>)+
   </SignedInfo>
   <SignatureValue>
  (<KeyInfo>)?
  (<Object ID?>)*
 </Signature>
```

The signed info contains information about the way the signature was made. The CanonicalizationMethod and SignatureMethod fields contains the algorithm used to perform the signature. The canonicalization methods is the structure used to create the document.

The reference field defines the file that is referenced. The transform value is the extraction algorithm to use.

The DigestMethod field is the method used to create the digest and the DigestValue contains its values. It is what bounds the user's key with the resource.

then the SignatureValue contains the signature of the document or of the part of document that has been signed.

### 6.2.2 Description of the XML signature solution for identification issues

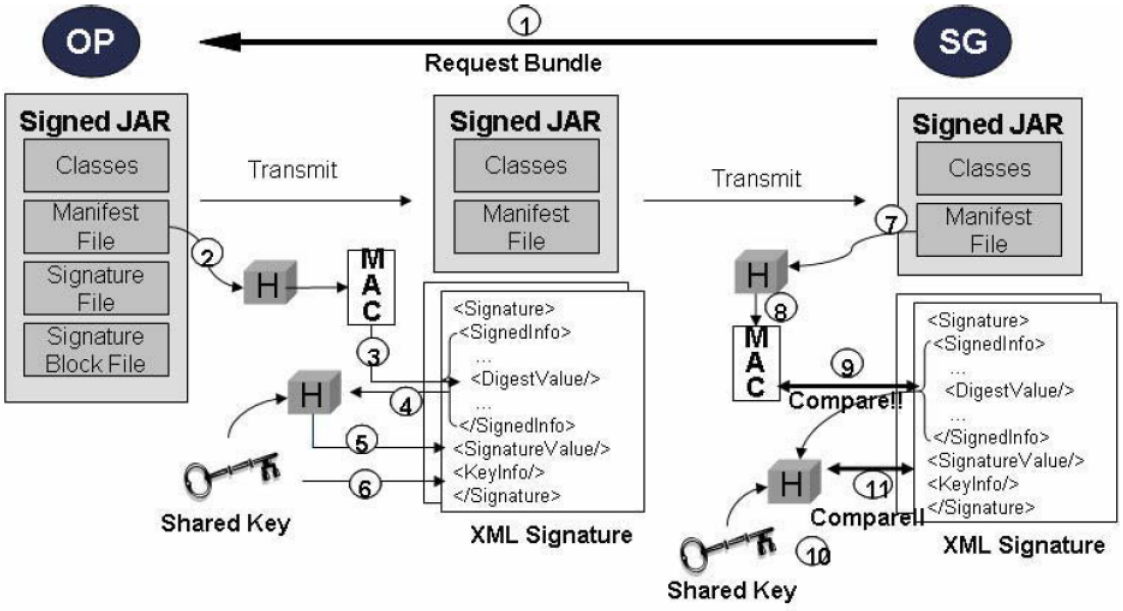


Figure 6.1: Bundle authentication using XML signature in OSGi platform [ea05]

1. when a bundle has to be transmitted, the operator contacts the service gateway which sends him the signed JAR file

2. The operator uses the algorithm referenced in the ¡Transform¿ field to make the signature of the manifest inside the JAR file. Then the reference element of the signature can reference the manifest directly.

3. The message authentication control is calculated by the operator and inserted in the ¡DigestValue¿ field of the signature. The algorithm used to calculate this MAC is stored in the ¡DigestMethod¿ field.

4. From the reference generation, the operator records in the ¡SignedInfo¿ element a signature and canonicalization method.

5. Using the shared key between the it and the service gateway, the operator crypts the XML signature located in ¡SignedInfo¿ in the ¡SignatureValue¿ element.

6. To authenticate the service bundle, the operator should provide information about the shared key. This information should be stored in the ¡KeyInfo¿ element.

   Then the JAR file with the attached signature is sent to the service gateway. There are two kinds of validations that are performed : the reference validation and the signature validation. The reference validation verifies that the contents referenced by ¡Reference¿ are unaltered and the signature validation verifies that the signature has not been altered.

7. the data stream to be digested is obtained by checking what is referenced by the signature (the ¡Reference¿ field).

8. The digest value of the signature is compared to the digest value obtained

9. The XML signature is compared with the signature info signed with the shared key the service gateway also possesses.

```
<Signature Id="MyFirstSignature"
xmlns='http://www.w3.org/2000/09/xmldsig#">
 <SignedInfo>
  <CanonicalizationMethod
    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
      0010315"/>
  <SignatureMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
  <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-
      20000126/">
   <Transforms>
   <Transform Algorithm="http://www.w3.org/TR/2001/
    REC-xml-c14n-20010315"/>
   </Transforms>
   <DigestMethod Algorithm="http://www.w3.org/2000/09/
    xmldsig#sha1"/>
   <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
  </Reference>
</SignedInfo>
 <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
 <KeyInfo>
  <KeyValue>
   <DSAKeyValue>
    <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
   </DSAKeyValue>
  </KeyValue>
 </KeyInfo>
</Signature>
```

Figure 6.2: XML signature of a service bundle [ea05]

To give the reader a better idea about a real signature, figure 6.2 represents an XML signature.

As we see, the users digested the manifest with a SHA-1 algorithm. They also used the C14N as a canonicalization algorithm. The digest message is set and the key is set but not completely visible.

### 6.2.3 Description of the XACML solution for authorisation issues

eXtensible Access Control Markup Language is an access control policies description language which represents policies with an XML schema.

This Policy system can specify the who, what, when and how information about a policy.

Our goal using such system is to grant access to resources or bundles to users. Such security is not needed for TURM as it is not a highly sensitive application (yet).

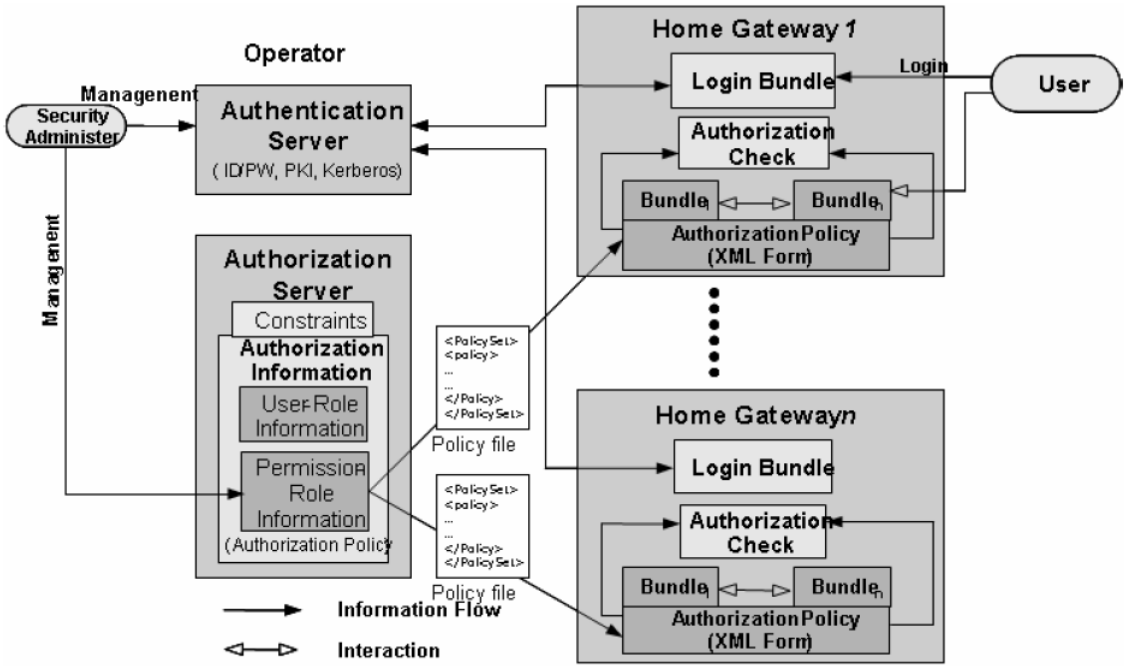


Figure 6.3: Authorisation architecture usin XACML

The operator is composed of an authentication server and of an authorization server. When a JAR file is transmitted, the operator not only creates the XML signature, it also generates and transmits the authorization policies relative to this bundle.

We must note that the policies are stored in the home gateway and that the only thing that the operator does is to identify the policies that can be applied to this user.

In this diagram, the User who authenticate himself can be regarded as a physical person. The object targeted by the policy can be one or some bundles and the permissions policies will generally be like "user X has access to bundle A, B and C". The permissions might also grant access to users to specific functions. These specific functions ideally located in bundles it comes down to the same thing.

Let's say you are in the developing department and that you use a plug-in application to manage the company you're in. If you need to pass order for new component needed for your job, you might not be able to command them yourselves because it will state that employees of the developing department are not permitted to access the "makeCommand()" operation which is located in the Financial_Commands bundle.

This provide thus an elegant way of managing policies, with the advantages and the shortcoming of each policies.

### 6.2.4 Benefits of the solution

Hee-Young Lim and Al.[ea05] made a table giving differences between OSGi's default authentication feature with their XML signature feature.

| Bundle authentication | Using RSH protocol | Using XML signature |
|---|---|---|
| Authentication | Y | Y |
| The scope of MAC processing | Every data transmitted from service to client | Manifest file included in service bundle |
| Encryption | Y | N |
| The size of file | Same with original size | Smaller than original size |
| Number of authenticated bundles | 1 | 1 and more |

Figure 6.4: Differences between RSA algotihm and XML signatures for authentication purpose

It is easy to notice that not only the XML signature provides a wider authentication than the RSA algorithm because it can authenticate more than one bundle, but also it uses a smaller amount of data because the bundles can be authenticated with only their manifests file and that with RSA you have to manipulate the whole bundle.

Again, these security operations are not needed for TURM because it does not have enough users and it is also not famous enough to be the target of hackers yet.

But as it is supposed to represent a reference implementation for policies expression languages maybe it might be a good idea just in order to provide a complete and robust implementation, to implement the security features.

## 6.3 Second solution : Aspect-Oriented Programming

Phu H. Phung and Al.[PHP08] described in their article a solution for OSGi which might be quite different: they try to define policy for each bundle using aspect oriented programming.

That's to say, they thought about writing a meta program that would control the bundles to block them whenever they can't access specific class or bundles.

To understand the solution, we must develop two concepts : aspect oriented programming and access control. In the access control concept, we will only explain one : RBAC.

### 6.3.1 Aspect oriented programming

Aspect oriented paradigm aims to separate the preoccupations in the code. Assume that we have an application to manage bank transactions. It needs to, of course, manage bank transaction but to add more safety, the management decides to implement logging features. A trivial way to implement these logging features would be to search the snippet of code that needs to be logged and if it was a perfectly coded application, it would not take a lot of time to do. Unfortunately, there are many applications that aren't suitable for such cross-cutting concerns. A cross cutting concerns is a modification that cannot be encapsulated or implemented in one or two classes. It is generally a modification that would affect the

whole code needing hours and hours of refactoring. To solve this more easily, aspect oriented programming defines a way to externalize this cross-cutting concern ( logging ) and to put it somewhere outside the source code in a autonomous way. [Pol04] This is what aspects oriented programming is for.

This is not a new way of writing code, it does not substitute to other paradigms, instead it works with them. Let's take back our example for further explanation. There are three steps to implement this solution:

- identify the key locations in the code where you want to implement the logging feature (defining join points)
- write the logging code
- compile the code and integrate it to the application

Here is an example of a join points for the example above:

```
pointcut employeeUpdates(Employee e):
call(public void Employee.update*Info()) && target(e);
pointcut employeeFinanceUpdates(Employee e) :
call (public void update*Info(Employee)) && args(e);
```

The first join point will be applied for each method call on the Employee class that will starts with update and ends with Info (the star has the same semantic as in regular expressions) and that has no parameters. The second join point will be applied for the same method call but which has a parameter Employee. This will also be applied for future developments if we decide to create another method to update the information of the employee.

Now that we have our join points, we must define what code should be implemented. This is called an advice In Java it works just as any other method except that it is placed in a new type called an aspect.

```
public aspect EmployeeChangeLogger {
pointcut employeeUpdates(Employee e) : call(
public void Employee.update*Info())
&& target(e);

pointcut employeeFinanceUpdates(Employee e) : call(
public void update*Info(Employee))
&& args(e);

after(Employee e) returning : employeeUpdates(e)
|| employeeFinanceUpdates(e) {
System.out.println("\t>Employee : " +e.getName() +
" has had a change ");
System.out.println("\t>Changed by " +
thisJoinPoint.getSignature());
}
}
```

Except for the code that is specific with aspectJ, this snippet looks like Java code: the class is implemented in a separate file and has the same structure. It works for the two pointcuts previously defined (due to the "or" in the third part of the snipped). This was feasible because the two pointcut have the same parameter. A choice of the author was to

execute the advice after the method call. There are two other ways to create an advice: before and around. While before is quite trivial to guess, around is used for example to replace the method's execution by the advice. When you use after, you can specify two types of behavior: after returning or after throwing if an exception can be thrown. The logging feature here prints out the fact that information about the employee was changed and his name. It was easy to do because the name of the employee is passed in as an argument to the advice. The second statement identifies the exact join point where the advice is executed and makes use of the AspectJ JoinPoint class. Whenever advice executes, it has an associated join point referenced by thisJoinPoint.

### 6.3.2 Acces control and RBAC (Role based access control)

#### Description

The role based access control (RBAC) is a higher level access control model used by most of the companies who own more than 500 employees. Instead of using groups, RBAC uses roles which are bound to specific permissions to perform certain operations. Generally a role is defined for a function in the company granting each employee of that function the same permissions. Even if groups can look like DAC groups, the difference lies in that groups consist of a collection of users while roles are a bridge between a collection of users and a collection of the Clark-Wilson model of transaction rights.

The Cark-Wilson model is an integrity model which defines a set of constraints. The data is in consistent or valid state if it satisfies the constraints.[Cla05]

RBAC is defined by three primary rules :

- Role assignment: A subject can exercise a permission only if the subject has selected or been assigned a role.
- Role authorization: A subject's active role must be authorized for the subject. With rule 1 above, this rule ensures that users can take on only roles for which they are authorized.
- Permission authorization: A subject can exercise a permission only if the permission is authorized for the subject's active role. With rules 1 and 2, this rule ensures that users can exercise only permissions for which they are authorized.

#### Benefits and Flaws

As RBAC is transaction oriented, the transaction rights help seeing which resources are accessed but also how they can be accessed. Also, as the policies are defined by role, it is much more easier for big companies to set the rights policies and to put the users in one role. Another asset of the RBAC model is the hierarchy management: each role can be allowed the permissions of a sub-role. RBAC also provides integrated support for the principle of least privilege, seperation of duties and central administration. The two first aren't supported by the MAC model and the third can be more or less implemented with trusted components but it is impossible to implement in DAC because it would violate the safety principle.

A shortcoming of RBAC is that as it has a fine-grained customized privileges, it is much more harder to manage all the roles for large systems. Also even if RBAC supports data abstraction through transactions, it cannot be used to ensure permissions on sequences of operations need to be controlled.

### 6.3.3 Abstract description of the solution

In order to control the access to certain bundles, we can set an RBAC policy whose subjects will be bundles. We will define interaction between bundles and which bundle they can access or not. Once this policy has been made, each time a service is registered, we might use aspect oriented programming to check at each operation call if the modified data could be modified by the bundle and if the accessed class could be accessed by the bundle. Then if it does not match the RBAC policy you can simply abort the operation with aspect oriented programming.

### 6.3.4 Interests of the AspectJ and RBAC coupling

This solution advantages are that it is easily deployed for someone who is familiar with aspectJ and RBAC. It is however expensive in the sense that every bundle will have to be modified with aspectJ code. The best case scenario would be an application like Turm : One core accessing services which do not interact with each other. In that scenario AspectJ only has to be implemented on the core and the external bundles can remain unmodified.

## 6.4 Third solution :Advanced OSGi security layer

In their paper, Chi-Chih huang and Al. [CCH07] aim to overcome four identified flaws of OSGi : system modification, invasion of privacy, denial of service and antagonism.

In this section we will first describe their architecture, then a case study and we will explain their conclusions.

### 6.4.1 Architecture of the advanced OSGi security layer

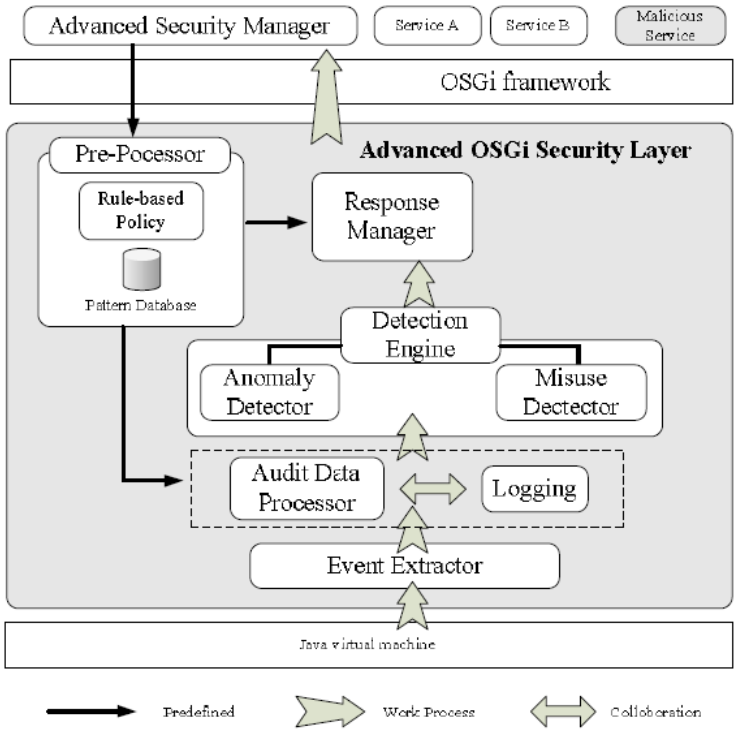


Figure 6.5: The model of the advanced security layer [CCH07]

AOSL is located between the JVM and OSGi framework. It can be administered by an administrator bundle which would configure the rules and monitor the status of services. In the pre-processor are located policies and rules.

The anomaly detector and the misuse detector will notice the AOSL whenever an illegal action is performed and the AOSL will then shutdown the responsible bundle.

The Event Extractor is the part of AOSL interacting with the JVM. It will notice the security layer whenever specific actions are performed. As it interacts with the JVM, it can get information about the program and about the system. The information gathered by the Event Extractor are Raw data and need to be manipulated by the Audit Data Processor in order to have sense.

The Audit Data Processor and the Logging Processor are working together to identify non-related security events such as an endless loop for example. Depending on how you designed the flaws detection algorithm, an even might be intercepted by the Audit Data Processor or go right to the Detection Engine. For instance : if the temperature of the processor considerably rises and that an application related process is having CPU time, then it might be on an endless loop.

The Detection Engine is made of two detector whose use have been discussed a lot in papers[AJ00] : the Anomaly Detector and the Misuse Detector. Those two detectors are responsible for handling events that are not already taken care of by the Audit Data Processor and the Logging Processor.

The Response Manager as its name said is responsible for the response the AOSL engages whenever there is a policy violation.

The Advanced Security Manager is simply a GUI which allows the user to monitor the status of the system, of the bundles and it also allows him to modify the detection algorithms or the policies.

### 6.4.2 A case study

Let's say we have three bundles : A B and C. B can access A's classes. Assume that B requires a sharable object from A. As it can access A, B get the sharable object. Then if B tries to share this object with C (which is forbidden to access A) it will just have to possess an operation which will use the shared object B retrieved.

With AOSL, every operations can be logged and observed. And every behavior can be analyzed. thus when C will try to get A, the operation will be logged and if a correct policy is set, the operation will be forbidden. As an instance of policy, we could put as a suspect behavior the fact that C invokes service B then invoke Service A. And we can set "ShutDown bundle C" as a response.

### 6.4.3 Evaluation

It seems like a robust and easy to use security layer, but the question of the performance remains. As we seem to have a strong security layer, we also need to not slow the application too much. Without communicating further details, Chi-Chih huang and Al. [CCH07] published a table relative to AOSL's performances.

## 6.5 Conclusion

In this chapter we described some solutions available in the field of security regarding OSGi applications. We saw they were not very complicated to understand or design, but that the difficulty remained in the actual implementation of the security feature.

We did not implement any in Turm because our manager's specification about security were very clear : they could be implemented if really needed and if that did not imply to

**Table 1. Results of the performance with or without the Advanced OSGi Security Layer**

| | 25 services | | 30 services | | 35 services | |
|---|---|---|---|---|---|---|
| | Execution time (ms) | | Execution time (ms) | | Execution time (ms) | |
| | Without AOSL | With AOSL | Without AOSL | With AOSL | Without AOSL | With AOSL |
| Httptest | 22.1 | 37.1 | 24 | 37 | 25.1 | 38.1 |
| URLhandler | 1 | 2 | 1 | 2 | 1 | 3 |
| SAX parsing | 28.1 | 31 | 29.1 | 35.1 | 29.1 | 37.1 |
| DOM parsing | 33.1 | 58.1 | 34.1 | 61.1 | 36.1 | 64.1 |

Httptest is a test case for OSGi HTTP service. URLhandler is a test case for OSGi URL handler service. SAX parsing is a test case for the OSGi XML Parser service using SAX mechanism. DOM parsing is a test case for the OSGi XML Parser service using DOM mechanism.

Figure 6.6: The performance table of AOSL with the author's comments. [CCH07]

rewrite all the code or to design a second application that the next developers will have to deal with and learn to use.

As all these solutions require some kind of "Meta application" we did thought as Turm is not yet a famous application used by thousands and thousands of users that it might not be needed. Furthermore, Turm as to be a reference implementation for an ODRL use, if we make it too complex, the reference implementation side of Turm will be harmed: the users who will inspect Turm's source code in order to understand to deal with ODRL licenses will also have to deal with the security layer with which the vast majority of developers is not familiar with. When it remains to Java code, the majority of developers who know Java will understand Turm easily.

But to remain complete, if a developer has to make a plug-in application for bank transactions or something with a high level of sensitivity, this chapter provides the basics of three solutions regarding the security for OSGi.

# Chapter 7

# Conclusion

## 7.1 Summary

In this Master's thesis, we identified the issues behind reusability of softwares. We realized that even if there were plug-in frameworks designed to help us code faster, they weren't well spread or even well kown. Some of them were pretty advanced like OSGi which developped a large community and is still very active nowadays, and some are less advanced because their purpose is not to make a robust application but more a quick sketch for a demonstration like JSPF.

We selected three frameworks and tried to learn their capabilities and how to use them. We identified some weaknesses in every of them. Some were minor like the fact that JSPF is a small community so that it might not survive to a huge growth of the users, some were major like the abandonment of JPF or at least of its main website.

We decided that OSGi was better because it was more powerful and that the documentation was a lot more spread. This will serve two objective : the easiness to learn for unexperienced programmers and the support the programmers will get as the community is huge.

Then we had to get a subject to make our experiment. We were given Turm which is a tool for usage rights management. Before defining the toolkit in itself, we first had to describe what it was made for, that's to say usage rights management. We explained that usage rights management was created in order to increase the awareness of the user about the media file they download. It gave them full information about the legality of their media file without enforcing them to behave legally. For this objective to be fulfilled, a community designed and is still improving a rights expression language : ODRL. This language is aimed to express policies for media files as permissions, duties and interdictions. It is also possible to specify an assigner which is the one who sells the license and the assignee which is the one who has to respect it. As a target one or more media files can be specified.

Once this was explained, we described Turm as a policy viewer and creator. This was its first purpose and then features were added to increase the user friendliness. For instance, it can extract meta data of a certain type of extension to show it conveniently to the user. Also as URM was about rights management, the creator of Turm also worked on a peer to peer interface in order to allow the user to exchange his media files with his relatives but with respect to the license files attached to them.

We then described in greater details where were the features implemented and how the application looked like with the source files. We identified some programming issues at the end to justify the fact that some clarifications weren't made in the code.

Then we had to justify the use of plug-in architecture for this software. As it should have a lot of extensions to manage, the use of plug-ins seemed a good choice. It also seemed a good choice because as Turm is a toolkit, new features could be developed for it and then

it might be convenient to represent these features as plug-ins to keep the application easy to customize.

We identified three ways of implementing a plug-in and explained that it was better to implement the external plug-in as a service and to keep the interface which can communicate with the plug-in in the core. In that way, as soon as the service is available, it register itself to the core plug-in's service tracker and can automatically be used.

The reason we did not share the main class with the plug-in is for obvious security reasons. Also it might be inelegant to share the whole program with a plug-in to "keep simple" because the plug-ins we designed did not need the whole core.

Then as we wanted to explain the limits of plug-in infrastructure, we decided to show some security weaknesses with the plug-in infrastructure and how some computer scientists proposed to solve them.

Some of them were really simple but had to be done for each plug-ins like the XML signature, and some of them were a bit more complicated and the explanations were quite incomplete like the OSGi security layer.

In the next part of the conclusion we will explain more clearly what contributions we made and what was already done before our thesis, we will explain again the issues we encountered during this thesis and will explain how we did solve them or why we did not solve them.

And as a final point, we will browse what is left to do in the fields our thesis used as a support.

## 7.2 Contributions of this thesis

### 7.2.1 Regarding Turm

Turm was not the main purpose of the thesis because we did not only describe Turm, but as it was the reference implementation we used for our tests, we had to get familiar with it.

The contribution we made within the application was a lot of cleaning work: we identified deprecated operations, classes, packages, and we left inside the program the ones that had been designed to be used by the peer to peer client currently under development.

We also rearranged the code by putting in the right packages the payload extractor's and the hash calculator's related classes because they were previously in the core.

We fixed path problems regarding (among others) hibernate and the ConfigurationManager class (which was put as a library and not as a source class in Turm).

Also, even if it might not be fully complete, the chapter about Turm might be a starting to a documentation which has been given up some time ago. By making a broad description of Turm, we have the beginning of a full class diagram because all the classes were represented with a class diagram (except the trivial ones). We also have a documentation which is more complete than a javadoc. As a javadoc is sometimes a more technical document, this chapter also explains what was in the mind of the authors and what were the clear links between the classes.

Last but not least, we implemented a plug-in infrastructure for Turm which is not huge, but will serve as a basis for further plug-in development. Not only we extracted the two features (payload extractor and hash calculator) but also we designed an easy yet robust and useful service tracker which will manage further implemented plug-ins. This work was also documented so new users can get familiar with plug-in development in general, and specifically for Turm. Evidently, we also commented the code we wrote regarding the plug-in architecture in order to the developer which is not familiar with OSGi to not be lost at first sight.

### 7.2.2 Regarding the plug-in programming

We did not reinvent what was already done. But what we did was an update to the plug-in development because JPF and JSPF still seemed a good solution if a user browsed google for an idea of plug-in framework. As JPF was dead and JSPF did not seem to be able to handle easily a large software package, we needed to explain what they could be used for, if they were still worth using according to our experience.

We did not set OSGi as the best programming plug-in infrastructure, but we did though identify it as an easy to install, easy to learn plug-in framework. This was an enlightenment because OSGi fulfilled both our clients requests and our personal and technical requests. As we said a lot in the chapter related to the framework comparison: OSGi may not be the best, but it is the most useful in our type of situation. That's to say, if someone doesn't really know how huge his software will be in several years, it is safer to take the most powerful tool than to take one that will be able to quickly solve his current issue but will hardly be able to handle a growth of the application's functionalities. Moreover in OSGi case, the fact that it has a lot of features does not affect the times it takes to learn to use it.

We also identified some security issues by browsing through the scientific papers already done and we explained what were the best known solutions to fulfill the security issues with a plug-in infrastructure. We also explained in which case it was useful: there is no use in assigning an IT team to a small program that will only run locally on one machine. It might be useful though to implement basic security with sensitive applications that might be the targets of hacking or denial of service attacks.

## 7.3 Difficulties encountered & solutions developed

### 7.3.1 Regarding Turm

The main issue we encountered was that the program was not maintained anymore in the sense that even if parts of it were implemented carefully, the whole application had not ran for a while and we thus had to handle the integration or re-integration of all the component to make it a small and working application.

The second issue was the lack of documentation for Turm. The solution here was simply to create one in this thesis by interviewing the authors of the different packages when we were uncertain about some operations or classes. We do not believe that our documentation will be the final version of it, but that we made the basic job in order to have an working and sufficiently documented version of Turm. Again, we did not write Turm's Javadoc (even if we commented many classes including ours), we rather focused on a higher level documentation to facilitate the comprehension of the new developers. This document would have spared us a lot of time we used on mails, conferences and appointments with the Koblenz team.

### 7.3.2 Regarding the plug-in programming

The biggest issue here was the documentation. OSGi was indeed broadly documented, but we had to pay attention to the fact that even in books, the documentation could be wrong. Some books included tutorials that were updated except for the source code which made the comprehension of OSGi quite difficult. Also, Many tutorials explained how to use OSGi with Felix, a command line framework. It was more convenient to use a graphical support which was provided by Equinox, the Eclipse OSGi plug-in.

The documentation was also a problem to compare the different frameworks. As JPF and JSPF were poorly documented, we could not have a full comprehension of these two

framework and that is why OSGi's section is longer than the two others. The lack of documentation also motivated us to choose OSGi for we would have more chances to find a quick solution to problems as we had more documentation (even if not fully trustworthy for the tutorials) and a bigger community eager to help us.

## 7.4 Perspectives

### 7.4.1 Regarding Turm

Regarding Turm and more broadly regarding user rights management, the possibility to make it a W3C standard is still not completed yet. Among others, an application using ODRL is required to convert it to a W3C standard. Also, there are other features that are currently being added: Florian Dietz is currently working on a service for Media players, P2P tools, etc. which might enable Turm to automatically know what he can and can't do with files. this might be a policy enforcement point start.

Writing of which, the usage rights management are just a theoretical design currently. There are no standard being worked on who might enforce user to behave legally. What is done actually is only information. There might be some researches to do in the enforcement field.

### 7.4.2 Regarding the plug-in programming

We did not do much theoretical progress in the plug-in programming. All we did was to inform the user of the existing solutions and what they were capable of. What might be left to do in the field is some good programming practices to write a plug-in.

There might also be some research to do regarding the plug-ins sharing system. The plug-ins are made to be reused, but (at least regarding OSGi) there are no "OSGi plug-in repositories" which might be a reference repository for OSGi bundles and which could be browed by users to find services they need. Something like websites where developers can find Jar files for eclipse might be a good idea [ser08] [jfc06].

# Bibliography


[AJ00] R.S. Sielken A.K. Jones. Computer system intrusion detection: A survey. *Computer Science Dept., University of Virginia*, 2000.

[CCH07] Ting-Wei Hou Chi-Chih Huang, Pang-Chieh Wang. Advanced osgi security layer. *Institute of Electrical and Electronics Engineers*, 2007.

[Cla05] Clark wilson integrity model. May 2005.

[CLM08] Vincent Englebert Claire Lobet-Maris. Introduction aux systèmes d'information : étude de cas. *FUNDP*, 2008.

[com99] OSGi community. Osgi official website. http://www.osgi.org/Main/HomePage, 1999.

[com08] W3C community. Secification of an xml signature's basic structure. `http://www.w3.org/2000/09/xmldsig#`, June 2008.

[com12] ODRL community. Odrl. `http://www.w3.org/community/odrl/`, April 2012.

[dCA11] Alexandre de Castro Alves. *OSGi In Depth*. Manning, 2011.

[ea05] Hee-Young Lim et al. Bundle authentication and authorization using xml security in the osgi service platform. 2005).

[gi95] Standish group international. Chaos report (1994). `http://www.ics-support.com/download/StandishGroup_CHAOSReport.pdf`, 1995.

[GJA08] Hongxin Hu Senior Member Jing Jin Gail-Joon Ahn, IEEE. Security-enhanced osgi service environments. *Institute of Electrical and Electronics Engineers*, 2008.

[HC02] Jean-Marie Favre Humberto Cervantes. Comparing javabeans and osgi towards an integration of two complementary component models. 2002.

[HLG10] Helge Hundacker, Verena Leisenfeld, and Rüdiger Grimm. A license aware peer to peer client with urm. *Virtual Goods*, September 2010.

[HPG09] Helge Hundacker, Daniel Pähler, and Rüdiger Grimm. Urm - usage rights management. *Virtual Goods*, September 2009.

[Ian01] Renato Ianella. Open digital rights management. Presentation at the W3C Digital Rights Management Workshop, 2001.

[IGPK] Renato Iannella, Susanne Guth, Daniel Pähler, and Andreas Kasten. Odrl wikipedia page. `http://fr.wikipedia.org/wiki/OSGi`.

[IGPK12] Renato Iannella, Susanne Guth, Daniel Pähler, and Andreas Kasten. Odrl version 2.0 core model. `http://www.w3.org/community/odrl/two/model/`, April 2012.

[jfc06] jar finder community. jar finder. {http://www.jarfinder.com/}, 2006.bibitem[JL09]fabric K. Vikram Xin Qi Lucas Waye Andrew C Myers Jed Liu, Michael D. George. Fabric :a platform for secure distributed computation and storage. *Institute of Electrical and Electronics Engineers*, 2009.

[JM10] Simon Archer Jeff McAffer, Paul VanderLei. *OSGi and Equinox: Creating Highly Modular Java*. Jeff McAffer, Erich Gamma, John Weigand, 2010.

[JPF] Jpf reference website. http://jpf.sourceforge.net/.

[Kri08] Peter Kriens. How osgi changed my life. *Association for Computing Machinery (ACM)*, January 2008.

[oc09] Stack overflow community. Java plugin framework choice. http://stackoverflow.com/questions/1613935/java-plugin-framework-choice, 2009.

[PHP08] David Sands Phu H. Phung. Security policy enforcement in the osgi framework unsing aspect-oriented programming. *Institute of Electrical and Electronics Engineers*, 2008.

[Pol04] Gary Pollice. A look at aspect oriented programming. 2004.

[ser08] serFISH.com. findjar. http://www.findjar.com/index.x, 2008.

[Vis10] Willem Visser. Jpf core system. 2010.